Netherlands organization for applied scientific research

TNO-report

TNO Physics and Electronics Laboratory

P O Box 96864
2509 JG The Hague
Oude Waalsdorperweg 63
The Hague, The Netherlands

Fax +31 70 328 09 61
Phone +31 70 326 42 21

report no. FEL-91-B166

copy no. 8

title

Voxel Data Processing on a Transputer Network

author(s):

Ir. W. Huiskamp

date :

June 1991

Original contains color plates: All DTIC reproductions will be in black and white

classification	
title	: unclassified
abstract	: unclassified
report text	: unclassified

no. of copies	: 30
no. of pages	: 118 (excl. RDP & distribution list)
appendices	: -

This document has been approved for public release and sale; its distribution is unlimited.

TNO

report no. : FEL-91-B-166
title : Voxel Data Processing on a Transputer Network

author(s) : Ir. W. Huiskamp
institute : TNO Physics and Electronics Laboratory

date : June 1991
NDRO no. :
no. in pow '91 : 708

Research supervised by : Ir. P.L.J. van Lieshout
Research carried out by : Ir. W. Huiskamp

ABSTRACT (UNCLASSIFIED)

With the growing availability of 3D scanning devices like Computer Tomographs (CT) or Confocal Laser Scanning Microscopes (CLSM) the need for high performance volume data (voxel) processing and display systems increased enormously. The recent development of a fast CLSM by IMW-TNO required a visualisation tool of matching performance. FEL-TNO was involved in the CLSM project because of its expertise in the area of fast visualization techniques using parallel processing. The FEL-TNO task in the project was to develop an experimental system that demonstrates the potential of parallel processing for volume rendering applications. This report describes the development, implementation and evaluation of the prototype 3D image processing system. Topics of the report are :

- introduction on volume data processing;
- introduction on Transputers and parallel processing;
- design of the Transputer based voxel processing system;
- implementation of parallel voxel visualization;
- implementation of parallel image processing algorithms;
- detailed description of the software;
- performance evaluation and scalability
- future developments.

to P. 119

rapport no.	:	FEL-91-B-166
titel	:	Voxel Data Processing on a Transputer Network
auteur(s)	:	Ir. W. Huiskamp
instituut	:	Fysisch en Elektronisch Laboratorium TNO
datum	:	juni 1991
hdo-opdr.no.	:	
no. in iwp '91	:	708
Onderzoek uitgevoerd o.l.v.	:	Ir. P.L.J. van Lieshout
Onderzoek uitgevoerd door	:	Ir. W. Huiskamp

SAMENVATTING (ONGERUBRICEERD)

Nu het aantal beschikbare scanners voor volume data (voxels) toeneemt is ook de behoefte aan krachtige verwerkings- en visualisatie systemen voor dit type data sterk gestegen. Voorbeelden van 3D scanners zijn Computer Tomografen (CT) en Confocaal Laser Scanning Microscopen (CLSM). De recente ontwikkeling van een snelle CLSM door het IMW-TNO instituut maakte een interactief visualisatie systeem noodzakelijk. FEL-TNO heeft een experimenteel systeem ontworpen waarmee de mogelijkheden van parallelle verwerking voor volume visualisatie worden aangetoond. De bij FEL-TNO aanwezige ervaring op het gebied van parallelle visualisatie technieken was de reden voor de samenwerking met IMW-TNO in het CLSM project. Dit rapport beschrijft de ontwikkeling, implementatie en evaluatie van het prototype. De volgende onderwerpen komen aan de orde :

-) inleiding volume data verwerking;
-) inleiding Transputers en parallel processing;
-) ontwerp van een voxel processing systeem met Transputers;
-) implementatie van parallelle voxel visualisatie;
-) implementatie van parallelle beeldverwerking;
-) gedetailleerde software beschrijving;
-) verwerkingssnelheid en schaalbaarheid;
-) verdere ontwikkeling.

INTRODUCTION AND EXECUTIVE SUMMARY

In this report the development, evaluation and implementation of an experimental parallel processing system for the processing of three dimensional datasets (voxel-images) is described. This type of system has applications in many areas :

-) Biomedical research for diagnostic purposes, surgery simulation, radiotherapy planning or anatomy study.
-) Materials research of geological samples or integrated circuits.
-) Computer vision, where RADAR or LASER range scanners record 3D data that is used for object recognition.

Chapter 1 gives an introduction on the subject of volume rendering and its applications. Sensors that provide suitable volume data are for example Computer Tomographs (CT), Confocal LASER Scanning Microscopes (CLSM) and LASER range finders. The drawback of voxel data rendering is that traditional computer systems need minutes or hours of processing time to compute a new image. The recent development of a fast CLSM by IMW-TNO required a visualisation tool of matching performance. FEL-TNO was involved in the CLSM project because of its expertise in the area of fast visualization employing parallel processing techniques. A list of the defined system requirements is given in the final paragraph of the first chapter. The main requirement is an interactive rendering performance (i.e. a result within 1 sec.) for a voxel dataset of 2 Meg.Byte

Chapter 2 describes the basic mathematical operations that must be performed to render a view on volume data from an arbitrary angle. The algorithms that are used to implement the different rendering options (e.g. front view, integration) are included also.

Chapter 3 gives an introduction on the terminology used in the field of parallel processing. The main features that make the Transputer a suitable node for parallel processing are discussed. The final paragraph presents possibilities to implement a parallel version of the voxel rendering algorithm. The choice for a distribution of voxel data slices accross the processor network is explained.

Chapter 4 discusses the parallel implementation of image processing algorithms into the voxel processor architecture. These operations are mainly intended as a preprocessing step for the voxel data (e.g. noise filtering).

Chapter 5 gives details of the developed pipeline architecture. Three main modules are discussed :

-) The Controller, that provides the user interface to the host computer (a PC-AT).

-) The Subcube Processors, where the actual rendering operations are executed. This code is running on many nodes in parallel.
-) The Graphics system that controls a display unit where the rendered images are shown. This module also functions as the interface to the 3D sensor (e.g. the CLSM).

The functionality of the units and the interface between them is defined.

Chapter 6 gives a detailed description of the software (OCCAM) that was written for the Voxel processor. This section is intended as software documentation for maintenance and development purposes.

Chapter 7 discusses performance and scalability issues. The required response time of less than one second can be achieved with 16 processors in the network. Higher rendering- and image processing performance is possible when more processors are added. This will require a (minor) change of topology into a tree structure. A typical result is that scalability is easier to achieve when the voxel dataset is large. This effect is explained by the overhead involved in starting up communication and computation in the processor network.

The last two chapters discuss future developments and conclusions. The potential of parallel processing for volume rendering applications has been clearly demonstrated by this project. The development of the prototype into a product will require a further improvement of rendering algorithms and the addition of more functionality. The system could be used as an accellerator attached to a (SUN) workstation. An X-window user interface via the workstation or dedicated Transputer hardware can be provided if desired. An important feature of the system is the flexibility to changes in resolution, performance and rendering algorithms. The voxel processor system is certainly not limited to CLSM images only, other sources of data are equally suitable (e.g. CT scans). The system will therefore be brought to the attention of potential users in other areas than Confocal Microscopy also.

1 PROBLEM DEFINITION

1.1 Introduction

The TNO Physics and Electronics Laboratory (TNO-FEL) in the Hague is part of TNO Defence Research. The activities of TNO-FEL focus primarily on operational research, information processing, communication and sensor systems. To support the fast data-processing usually required in sensor system applications, research was started on parallel processing. This research has now resulted in two major application areas : real-time computer generated imagery and 3D image analysis, processing and visualization. In this report the development, evaluation and implementation of an experimental parallel processing system for the processing of three dimensional voxel-images is described. This type of system has applications in many areas :

-) Biomedical research, the traditional use was for diagnostic purposes only, but in recent years surgery simulation, radiotherapy planning and anatomy study have also become important. In some systems the visualization system has been combined with numerically controlled molding machines to construct models of the voxel data for use as implants or to practise surgery on.
-) Materials researchers apply CT and CLSM scanners to investigate natural or artificial material like geological samples or integrated circuits.
-) A new application area is computer vision. RADAR or LASER range scanners are used to record 3D data. This data is preprocessed and used for object recognition to control industrial production or to guide an autonomous mobile system (a robot).

The topics covered in this report are :

-) voxel data visualization.
-) parallel processing and Transputers.
-) distribution and communication concepts.
-) implementation of parallel voxel visualization algorithms.
-) parallel implementation of statistical computations, image enhancement and image analysis operations.
-) performance evaluation, scalability.

1.2 History

The invention of the X-ray system by Wilhelm Rontgen in 1895 marked the start of electronic imaging for medical diagnostics. During the next 70 years the technology was significantly improved, providing better images with lower radiation levels. The X-ray images however represent only a 'shadow image' of the three dimensional object : rays are sent through the sample and are absorbed differently, depending on the encountered material. This means that the three dimensional structure of the object is lost in the resulting image and that a radiologist needs many years of experience before being able to interpret the image correctly. In recent years, new X-ray systems have been developed which are free of the superpositioning of the contributions of all the material in the sample. This system is called Computer Tomograpy. A Tomograph collects data of a cross-section (Tomogram) of the sample with a rotating X-ray. This data can not be directly interpreted as an image, it needs extensive processing first (like Fourier Transforms). The result of this processing step is the X-ray image of the sample cross-section. The whole process can be repeated for many different cross-sections of the object, resulting in a 3D dataset. The radiologist can now 'scan' through a stack of cross-sections and draw conclusions about the position and shape of objects within the dataset. It is however still necessary to extract 3D information from a series of 2D images. This can be very difficult for some objects. Furthermore, it would be useful to be able to look at the data from angles other than the one it was recorded in. The reason for this is that some data may be hard to interpret from the given direction. A more recent type of sensor (ca. 1975) that provides 2D cross-sections of a 3D object is an NMR scanner (Nuclear Magnetic Resonance), which is particularly suited to show soft tissue, like skin, blood vessels etc. This sensor also needs extensive pre-processing before the basic 2D data is available, and again the 2D data can be combined to build up a 3D dataset. Another source of volume data is provided by the Confocal LASER scanning microscopes (CLSM). This type of microscope differs from the conventional microscopes by the extreme depth discrimination.

With the growing availability of 3D scanning devices, the need for high performance image processing and display systems has also increased. The main purpose of these systems is the visualization of the (unknown) object in such a way that its spatial structure can be understood. An additional demand is that the system is fast enough to be used interactively.

1.3 The Voxel Data

In 2D image processing the samples that form an image are known as picture element or 'pixels'. Since in our case the 2D images are cross-sections of a 3D dataset the basic samples are called volume elements or 'voxels' (Fig. 1.1). Volume images are normally represented as a series of parallel two dimensional slices (Fig. 1.2). These slices can be obtained from a range of possible sensor systems, examples are Computer Tomography (CT), Nuclear Magnetic Resonance (NMR) or Confocal LASER Scanning Microscopy (CLSM). The department for Electro-Optical Systems of TNO Institute of Environmental Sciences (IMW-TNO) has recently developed a new type of CLSM [1]. The main feature of this system the ability to record images in near real-time. This made it important to have a visualization system that matches the CLSM's speed. The operation of a CLSM is based on the illumination of an object by a LASER beam. The reflected beam is projected on a spatial filter (e.g. a pinhole), so that only light coming from the focal point of the LASER is measured (Fig. 1.3). TNO-FEL was involved in the CLSM project because of its expertise in the area of fast visualization techniques using parallel processing.

Voxel representations are not only very suitable for applications with 3D empirical data. It can also be used with synthetic data, for example solid modelling or fluid dynamics simulations. Before volume rendering became feasible, experts had to interpret the slices to deduce the 3D information. Until recently, computer assisted techniques to visualize the volumes interactively were based on displaying contours only, because of the processing time involved. These contours often had to be traced manually from the actual data. Full use of the 3D data could only be made through off-line computing. Because of the large number of voxels involved, a considerable processing capacity is required. Several architectures based on dedicated hardware have been proposed to increase performance [2], [3]. Such a dedicated system however has the disadvantage of inflexibility to any change in rendering options or object sizes (also the cost is high).

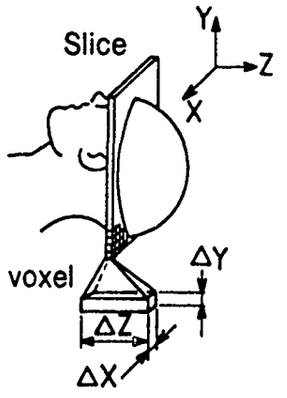


Fig. 1.1 Voxel definition.

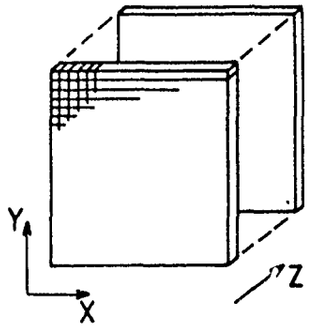


Fig. 1.2 Voxel data set

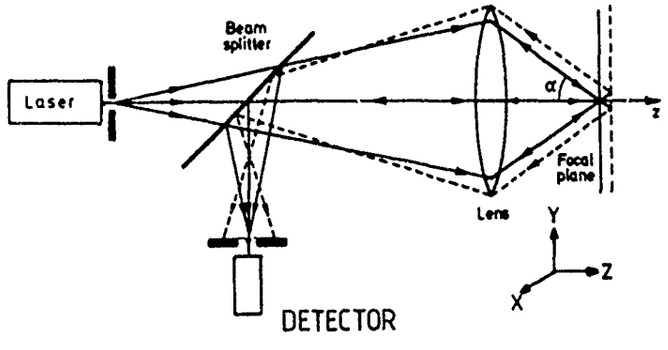


Fig. 1.3 CLSM operation principle

1.4 System requirements

The following list of requirements for the voxel-processor was defined after literature study and discussions with the CLSM developers from IWM-TNO:

a) Voxel data-set rendering from an arbitrary angle.

b) Provide an interactive rendering speed (< 1 sec.) for a voxel set of 128*128*128 (or any other set of 2 Mega voxels). Each voxel is represented by a greyvalue between 0..255, i.e. a single byte.

c) Use a parallel projection method, no perspective distortion is required for this prototype.

d) Voxels are expected to be cubic. This means that the resolution of the scanner is identical in x, y and z direction. In practice, scanners do not allways have this property. The problem is usually solved by resampling (interpolating) the voxel image.

e) Selection of a 'Volume-Of-Interest' within the available voxel data. Through this option uninteresting or disturbing parts of the voxel-image may be 'peeled away'.

f) Selection of a cutting plane through the object; voxels in front of this plane will not be visualised. This option will create a cross-section or 'Z-cut' through the object after rotation (Fig. 1.4).

g) Allow the selection of a threshold; voxels with a value below this threshold will become transparent.

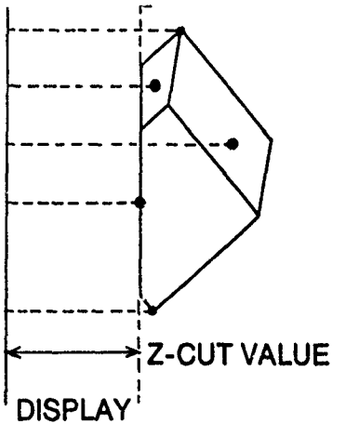


Fig. 1.4 Voxel data cross-section

h) Provide a set of basic image processing operations that can be applied on the voxel data before visualization. The operations must include noise reduction and edge detection methods.

i) Support editing and selection of different colour look-up tables. This feature enables the use of pseudo-colours or grey-scale transforms for certain intensity values, thereby increasing the visibility of interesting areas.

Several ways of rendering the transformed data on the screen are possible, the currently required options are :

a) front view

Display the object's intensity, as seen from the selected orientation (Fig. 1.5).

b) depth shading

Display the object's 'distance' from the screen at each location, resulting in a realistic depth illusion (Fig. 1.6).

c) integrate function

Display the object's density at each screen location (Fig. 1.7).

d) layer view

Display an intensity related to the layer from which the visible voxel originated (Fig. 1.8).

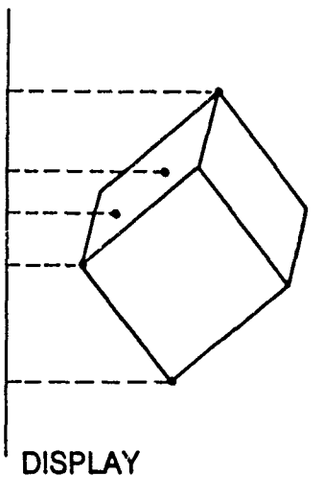


Fig. 1.5 Front view

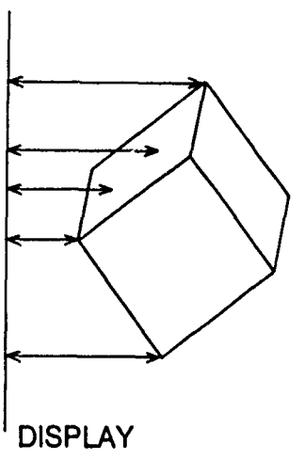


Fig. 1.6 Depth shading

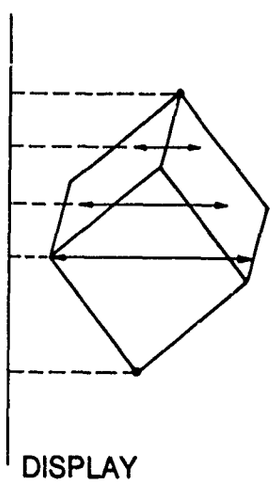


Fig. 1.7 Integrate function

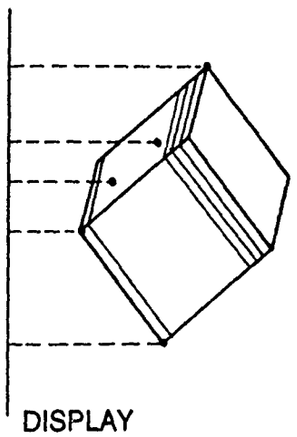


Fig. 1.8 Layer view

Some additional system design requirements were defined also :

-) The necessary computing power will be provided without the use of dedicated hardware. This will be achieved by processing the data in parallel on a network of Transputers. Previous work by others had shown that Transputers can be used effectively for medical imaging applications [4].
-) The system will use a PC-AT as a host computer, so it does not need to be completely standalone.
-) Voxel-data sizes depend largely on the sensor type, in CT scans for example it is possible to get resolutions of 512*512*128 with 12 bits per voxel. The developed system should be modular in its construction, both on a hardware and on a software level, so that it can deal with varying size- and performance- demands. This implies that system should be flexible to any change in rendering options or object sizes.
-) The system should have a scalable performance, this implies that its cost/performance ratio is flexible.

2 VOXEL VISUALISATION

2.1 The 3D Transformation

The Voxel data is represented by a cube in 3D space (Fig. 1.2). Displaying this data under different angles on a 2D screen involves a 3D transformation of the object space to the display space (i.e. the observer coordinate system), this is represented in fig. 2.1. The carthesian coordinates of the object space are chosen to correspond to the x, y and z indices of the voxel data set. This implies that the voxels are considered to be cubic. The transformation from the object space to the display space (fig. 2.2) consists of a vector-matrix multiplication on each voxel coordinate (i.e. a vector).

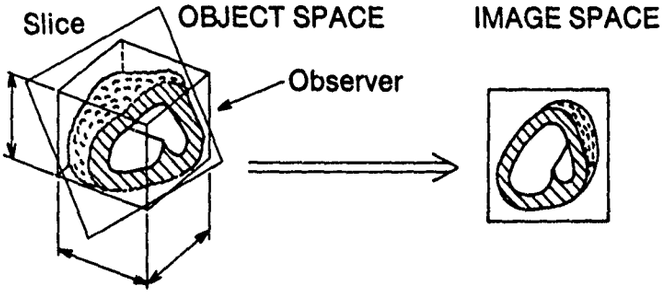


Fig. 2.1 Object space to display space transform

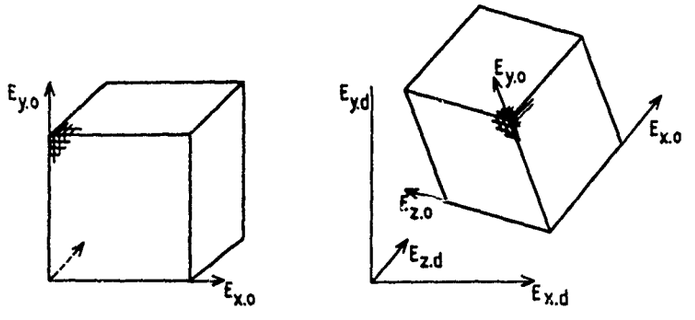


Fig. 2.2 Object and display coordinate systems

The rotation result of a two dimensional vector about an angle 'a' is illustrated in fig. 2.3. The new coordinates (x', y') of point (x, y) are given by :

$x' = x*\cos a - y*\sin a$

$y' = x*\sin a + y*\cos a$

An equivalent set of relations can be derived for rotations in a three dimensional space.

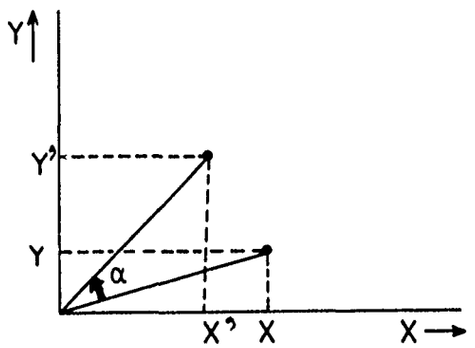


Fig. 2.3 Rotation over 'a' about origin

The object transformation matrix is formed from a combination of the matrices for rotations about the X-, Y-, and Z- axis :

$$R = R_z * R_y * R_x =$$

$$= \begin{vmatrix} (cY*cZ) & (sX*sY*cZ-cX*sZ) & (cX*sY*cZ + sX*sZ) \\ (cY*sZ) & (sX*sY*sZ+cX*cZ) & (cY*sY*sZ - sX*cZ) \\ -sY & (sX*cY) & (cX*cY) \end{vmatrix}$$

('c' for cos, 's' for sin)

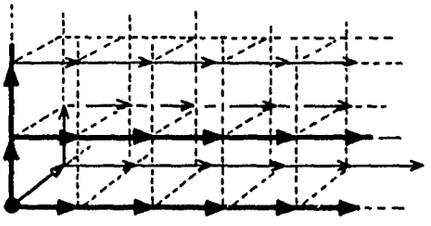
(X, Y, Z are rotation angles about x-, y-, z- axis).

Since vector-matrix multiplication is a linear operation and all voxel coordinates have to be transformed, it is not necessary to perform this multiplication for each coordinate. We may instead use three simple additions to step from one transformed coordinate to the next (Fig. 2.4). This method offers a considerable reduction in computational load :

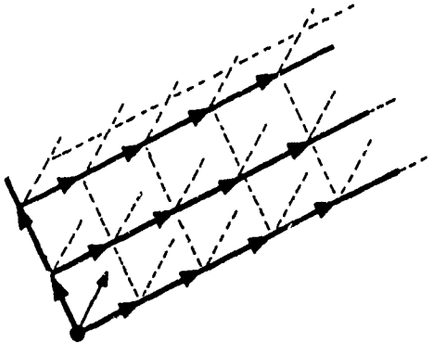
-) Vector-matrix transform (9 multiplications and 6 additions) requires about 20 us per coordinate.

-) Incremental transform (3 additions) costs less than 1 us per transformed coordinate.

The incremental steps along the E_o_x, E_o_y or E_o_z axis of the voxel data (object space) correspond to incremental steps in the display space (E_d_x, E_d_y, E_d_z). A unit step in x axis direction will result in the addition of the corresponding transformed (1,0,0) unit vector to the previously computed coordinate.



UNIT STEPS IN OBJECT SPACE



UNIT STEPS IN DISPLAY SPACE

Fig. 2.4 Incremental object transform

2 2 The 3D projection

The second important part of the visualization process is the projection of the transformed 3D data onto a 2D surface (the screen). The projection operation involves several tasks :

1) Perspective correction.

   The perspective correction is not required for this system, given the fact that a parallel projection was prefered. The advantage is that measurement of voxel distances can then be made from the screen. The disadvantage is that the image on the screen will show some perspective distortions. Addition of this operation would involve the division of the transformed x and y voxel coordinates by its z coordinate to obtain the correct perspective projection of all voxels on the operator display. The prototype uses the uncorrected x and y values for this projection.

2) Clipping of data that is not visible from the given eyepoint. In this implementation there will be no need for clipping, because 'zooming' is not supported and the screen size is chosen sufficiently large for the resulting image to fit on it for all voxel-data orientations. However clipping must be used when larger voxel-data sets are employed without increasing the screen resolution. The computational cost consists of two comparisons for the x and y coordinates of each transformed voxel against the given screen dimensions.

3) The hidden surface elimination : 'distant' voxels are obscured by 'closer' voxels if they are projected onto the same location on the screen (Fig. 2.5). There are several methods to achieve this goal :

   -) The Z-buffer algorithm compares the z-value (the distance to the viewpoint) of a new voxel projection with the z-value of the voxel that was previously projected onto the given screen location. The Z-values for each screen pixel must be stored and updated whenever a pixel value changes. This method is often used because the dataset can be traversed in a random order.

   -) The Painters algorithm avoids depth comparisons per pixel by traversing the voxel-data in a back-to-front direction. When generating the screen this way, new pixels can simply overwrite any old value. This method is faster than the Z-buffer algorithm and it uses less memory but the disadvantage is the need for depth sorted data.

   The second method is selected for the voxel processor since it is the most efficient method, given the fact that voxel data is automatically stored in its geometrical order at the time of recording.

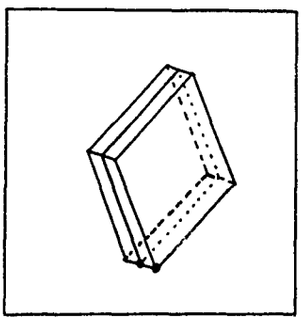


Fig. 2.5 Hidden surface elimination

4) The actual rendering mode of the voxel data is an integrated part of the 3D projection. The possible modes are implemented as follows :

   a) Front view.

      The original grey value of a voxel is mapped on the screen projection of the transformed coordinates.

   b) Depth shading

      The Z-value of the transformed coordinate is mapped on the screen. This implies that the actual Z-value must be scaled down to prevent it from becoming larger than 255, which is the maximum allowed grey value.

   c) Integrate function

      The value on the screen represents a count of the number of voxels that projected on that specific screen location. This value must first be initialised to zero and it must be clipped to a maximum of 255 during the computation.

   d) Layer view

      The screen value represents the layer number (i.e. the z coordinate) from which the projected voxel originated.

   The projection process will perform two additional tests on the voxels before they are used for the rendering operation :

   1) The Z-value is tested against the cutting plane, voxels in front of this plane are not visualised.

   2) Voxels with a grey value below the user selected threshold are not considered for further processing.

The center of the voxel data set will be positioned in the center of the projection screen for all possible orientations. This is achieved by giving a certain offset to the transformed origin of the voxel data. The relative position of the transformed voxel data set in display space is shown in Fig. 2.6. Data set dimensions are (DX*DY*DZ) and the resulting occupied display space is a cube with dimensions (ID*ID*ID). The resulting image is computed as the projection of the transformed voxel set on the frontal face of the (ID*ID*ID) cube.

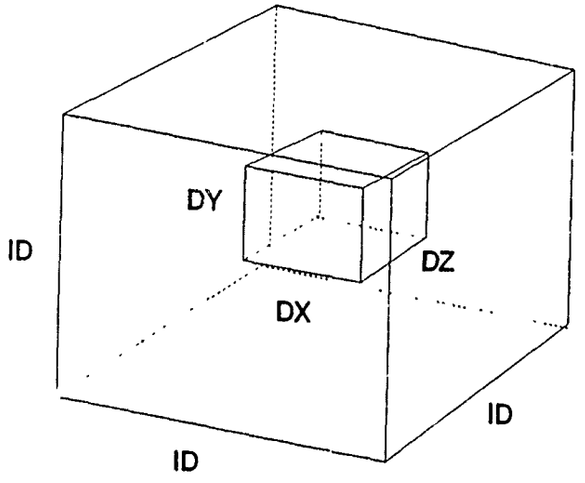


Fig. 2.6 Voxel position in display space

2.3 Supporting systems

Next to the main tasks that were described in the previous sections, some additional support must be added to the voxel processor to construct a fully functional system :

1) A controller process that provides :
   - -) Interface to the host-PC to supply keyboard, screen and file i/o.
   - -) Menu based user interface to control the voxel processor.
   - -) Debug and error reporting.
2) A means to display the resulting images in grey values or pseudo-colors. The resolution of this display unit is chosen to conform to CCIR TV standards (512 * 780 pixels) for the following reasons :
   - -) The CLSM uses this format.
   - -) Commercially available hardware can be used to both 'grab' new data from the CLSM and display results from the voxel processor.
   - -) Normal VCR systems can record the results from the system for play-back at a later time.

The dimensions of the voxel data set have a direct relation with the resolution (ID*ID) of the graphics unit when displaying without clipping or scaling is required. Fig. 2.7 shows this relation, a calculation proves that a 256*256*32 transformed voxel set can be displayed from any angle onto the 512*780 screen.

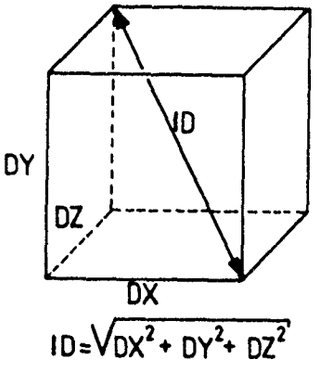


Fig. 2.7 Image dimensions

3 PARALLEL PROCESSING

3.1 Computer Developments

Computer applications tend to need increasing amounts of processing capacities. Single processor systems are reaching the limits of performance improvements. It is obvious that using more processors running in parallel could (theoretically) provide unlimited power. Most existing multi-processor systems use a common communications channel (the bus) for interconnections. With a growing number of processors the bus capacity becomes a bottle-neck for system performance. Communication bandwidth of the network must be increased also when processors are added. Providing processors with direct connections for all data exchange will supply this increased bandwidth.

Several classes of multi-processor systems have been defined [5] :

-) Single Instruction Multiple Data (SIMD). Each processor in the network will execute the same instruction (synchronously) on different data. Array processors fall in this class. Examples are image processing applications where each processor performs the same filter operation on a different part of the image.
-) Multiple Instruction Multiple Data (MIMD). Processors can all be running different programs, sending results to others when they are finished. Examples are pipelined systems or multi-user applications.

Many existing sequential programs could benefit from being able to perform more than one action at a time. It is however generally not trivial to implement a parallel program on a processor network. Problems arising are :

-) Decomposing the problem in a number of processes running in parallel.
-) Allocate processes to processors and select the network topology.
-) Load-balancing the processors.
-) Distributing data across the processors.
-) Efficient inter-processor communication.
-) Synchronization between processors.
-) Debugging the software.

3.2 Transputers

The INMOS T800 Transputer is a computer-on-a-chip, containing a 32 bit RISC ALU, a 64 bit Floating-Point Unit, memory and four high-speed (1.5 MByte/s) input/output links for point-to-point communication (Fig. 3.1, [6]). The Transputer was designed for efficient parallel processing : it is a high performance component (10 MIPS, 1.5 MFLOPS), with an on-chip process scheduler and low-overhead communication facilities. A network of Transputers may be constructed by connecting them via links (Fig. 3.2). Each Transputer in a network has (private) local memory to store program and data. Transputers may be programmed in high level languages like PASCAL, FORTRAN or C. These languages must have facilities added to implement the special features of the Transputer (processes running in parallel, communication etc.). OCCAM is a language that was developed by INMOS to describe parallel processing and communication via channels [7]. In fact the Transputer may be considered a hardware implementation of OCCAM. Transputer versions without the floating-point unit are also available : the T424 (32 bit) and the T222 (16 bit).

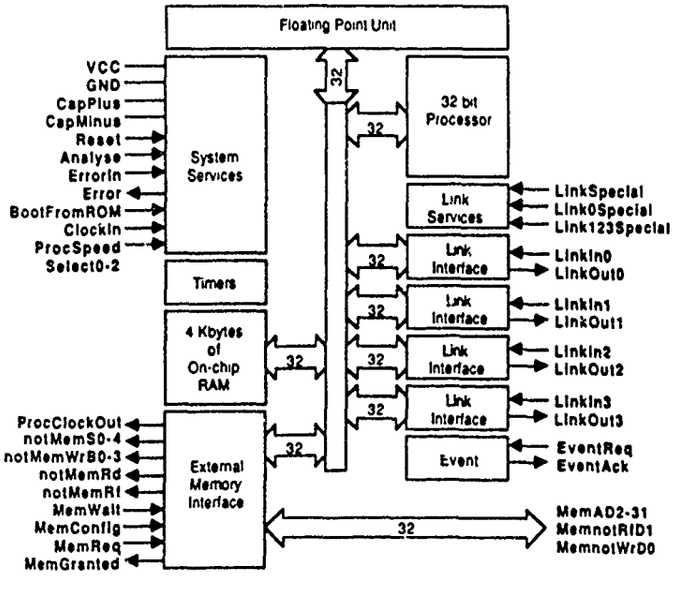


Fig. 3.1 T800 Block Diagram.

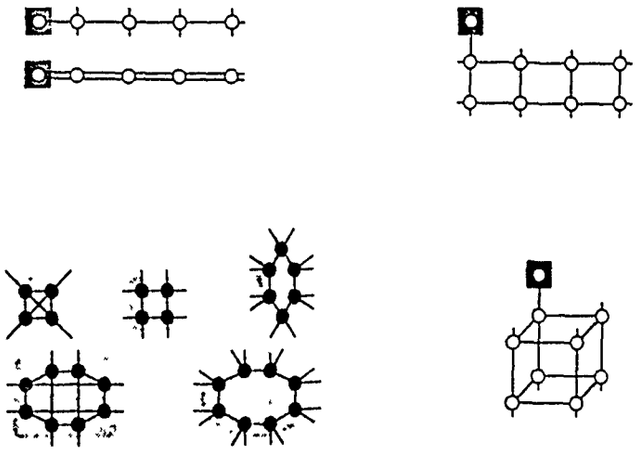


Fig. 3.2 Transputer networks

Transputer networks belong to the MIMD class of parallel processing systems, all nodes in a network are basically independent units, communicating and synchronising only when necessary. An MIMD network is the most flexible solution to parallel processing, since part of the network may actually be operating as SIMD.

At TNO-FEL, research has concentrated on the Transputer as the computational element in parallel processing applications, because of its useful features, high performance and software support. This explains the reason for TNO-FEL to apply a system of these programmable (low cost) processors operating in parallel for the implementation of the voxel processor prototype.

3.3 Problem decomposition

The difficulty with parallel processing is to find an effective way of decomposing the problem in a number of processes that can run concurrently. There are two basic approaches to this problem :

A) Data parallelism : split up the data in independent parts. Each processor in the network will essentially perform the same operation on a different part of the data set. This option implies that the original (sequential) algorithms may be used and that it is rather straightforward to add more processors to the system. Load-balancing is in general easy, as a consequence of using the same code on all processors.

B) Algorithmic parallelism : split the algorithm in several parts and assign these parts to different processors. This option will often force you to re-design the algorithm, since automatic extraction of parallelism in program code is (at this moment) hardly possible. Adding more processors and keeping a good load-balance is not trivial, when using this type of parallelism. In general, we found that this approach is only effective if the problem is computation-bound rather than data-bound.

The data parallelism option was chosen in our system, since voxel-visualisation is a data-bound problem, the actual algorithms involved are not that computationally intensive. A second reason is that this solution provides better scalability. Data parallelism may be accomplished in (3D) image processing by splitting up the computations in display-space or in data-space :

-) Display space parallelism implies that each processor is assigned to a certain area of the resulting image (e.g. a number of scanlines). Since views of the rotated voxel-image will be generated, this solution implies that each processor must have access to the complete voxel-image. Complete access is possible when a voxel-image copy is stored in each processor (large memory requirement) or alternatively, processors could request voxel-data elements from a central store, when needed (communication overhead). Load-balancing may be a problem, since the most computation intensive parts in the display-space will shift according to the rotation angle. Ray-tracing is a typical example where parallel processing in display space is often used. The load-balancing problem can be tackled by implementing a processor farm. In this construction a controller process 'farms out' a new piece of work (i.e. a part of the display) to each processor in the network as soon as it has finished work on a previous part. The controller does not need to know which processor will actually perform the job.

-) Data space parallelism is based on access of a limited part of the original voxel-image. This implies that each node is assigned to a section of the voxel-image, which is stored locally. A node will produce the contribution of the local data to the result. The actual result will be available after combining (merging) all the contributions.

The advantages of the second method over the previous one are :

-) Less memory requirement.
-) Fast access to the (local) voxel-data.
-) Good load-balancing, all contributions will need the same computation time, when the voxel-data sizes are equal.

Disadvantages are :

-) The overhead of the merging operation
-) Some data is calculated by the nodes that is not needed in the final result.

Typically, a large number of views will be generated from a single (large) dataset. Therefore, the lower communication need of the second method was the reason to choose data-space parallelism in the voxel-processor system. Each Transputer is holding a data-segment which is a unique part of the complete voxel set. This data-segment could be formed in several ways :

-) A segment consists of a cross-section through a number of 2D slices (Fig. 3.3).
-) A number of complete slices could be assigned to each processor (Fig. 3.4).

The slice based distribution method is selected for implementation, the reasons are :

-) Using slices will give maximum values for data-segment sizes in two directions. This implies that all voxel processing (visualization, merging, displaying etc.), which is executing in program 'loops', will have maximum runs in these two directions. In general it is more efficient on computers to keep 'loops' running as long as possible.
-) Image processing operations are also planned to be part of the system and these algorithms are generally based on 2D images. The slices are interpreted as 2D images, since voxel objects normally have a higher resolution in X and Y than in Z direction.
-) Voxel data is recorded on a slice basis, this makes it the most natural way to process it.

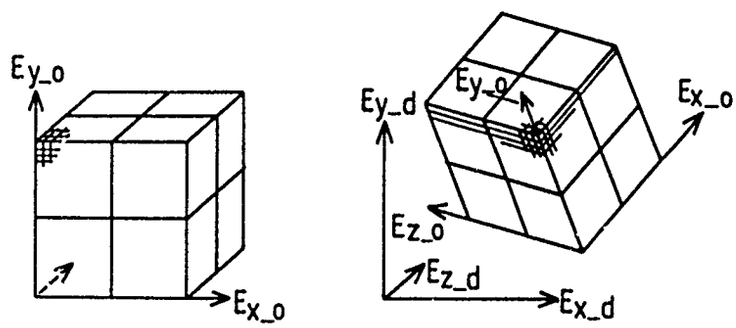


Fig. 3.3 Block distribution.

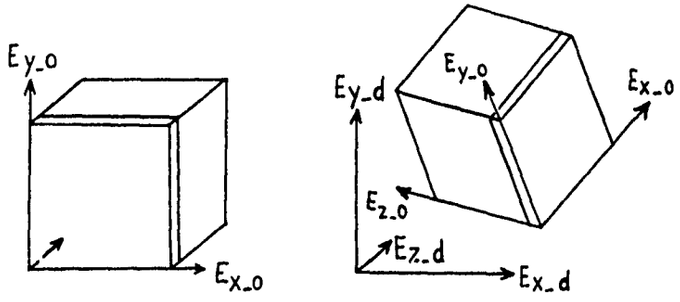


Fig. 3.4 Slice distribution.

The merging process combines partial results from all segments. Each partial result is added to the temporary result on a 'back-to-front' basis. This implies that the partial result (sub image) contributed from a single segment can be readily added to the partial result of its direct neighbour when neighbouring processors have geometrically neighbouring slice data (Fig. 3.5). The geometric order of the data set must be preserved in the network architecture. The merging process could not be performed locally in the network if this order is not supported. It would then become necessary to transmit all partial results to a central point where the merging could be performed in the correct order. In our architecture this geometric order is given. In order to combine the two partial results in a correct way, the merger needs some additional data : the subcube's priority. The priority is based on the z-value of the subcube's transformed origin. The lowest priority is for the sub-cube with the largest distance from the viewer. The partial results of this subcube will be obscured by any sub-cube result of a higher priority. Figure 2.5 and 3.5 illustrate this process.

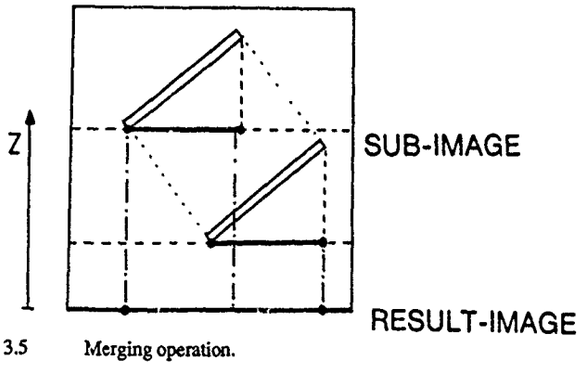


Fig. 3.5 Merging operation.

4 IMAGE PROCESSING

4.1 Introduction

The original voxel images from scanning devices tend to be noisy in many cases, so noise filters are often needed. Some of these operations have been implemented in a prototype image processing system designed for the CLSM by IMW-TNO [8]. When voxel images are recorded with the voxel processor system's build-in framegrabber, some preprocessing (i.e. averaging) can be done on this board before sending the slices to the processing elements. Further image analysis operations (edge detectors etc.) are also required as an integral part of the voxel processor to provide a useful system.

4.2 Parallel image processing

Most low level image processing operations are well suited for arrays of SIMD machines, while higher level operations can be implemented more effectively on MIMD systems. Research into the use of Transputers for image processing has shown that processing speed can be increased very linearly for non real-time applications [9], [10]. The limited Link bandwidth will however present a serious problem for the distribution of input images and the collecting of partial results for real-time applications. In this case, it will become necessary to implement a hardware bus system for the data i/o. This problem has been identified and is being tackled by several computer manufacturers at the moment. The aim of our project is to develop a flexible and modular architecture, suited for both visualization and processing of 3D datasets. Dedicated hardware for image processing will have better performance than programmable systems, but it lacks the flexibility to meet changing demands. The image processing operations were therefore fully software implemented on Transputers. Parallelism for image processing can be achieved in two ways : data-parallelism and algorithmic parallelism. Since data-parallelism has been chosen for the visualization, it is obvious that each Transputer will also perform the image processing on its local data.

4.3 Implementation

For many (2D) image processing operations, a new pixel intensity (g) is computed based on a weighted sum (h) of its own original value (f) and the values of its nearest neighbours :

$$g(x,y) = \sum_{p=0}^{n-1} \sum_{q=0}^{n-1} h(p,q) \cdot f(x+p,y+q)$$

This operation presents a problem when the pixel data is distributed across several processors : the neighbour values may be located on a different Transputer. There are two basic solutions :

- -) Communicate pixel values between neighbours, (the data swapping may be done in parallel for many Transputers). This method puts additional demands on the network topology and requires some communication facilities.
- -) Provide data overlap between neighbours. In this case each node needs additional memory. The amount of extra memory depends on the dimensions of the convolution matrix h and the size of the slices.

A second problem arises on the picture edges, since these pixels have no neighbours. Simple solutions are to assign zero values to the non-existing neighbours or to wrap-around to the other side of the image.

The remarks given above are valid for 2D images distributed across a Transputer network. In our application we deal with 3D datasets, with a distribution based on 2D slices. The image processing operations must therefore also be extended to 3D. This implies that 'neighbour pixels' may now be located in different slices. It is however in many cases possible to use normal 2D operations within a single slice. This is caused by the relatively low correlation between voxels in adjacent slices : most scanners have a much higher resolution in x,y direction than along their z-axis. This lower axial correlation in the data was one of the reasons to choose slices as a basis for data-parallelism.

Presently, our system uses overlapping data-slices between neighbour Transputers because of simplicity. In future versions (when the size of the filters exceeds 3*3*3 or when the datasets become larger), data communication will be implemented. The selected network topology is suited for this method.

Local 3D image processing algorithms are based on their 2D counterparts [11] and operate on a (3*3*3) space). Currently implemented operations are :

-) Mean filter.
   The mean value of the (3*3*3) neighbourhood of a voxel is assigned to it.
-) Laplace filter.
   The maximum value of three orthogonal 2D Laplace convolutions is assigned to the voxel at the center of the Laplace transform. The result is the detection of high spatial frequencies in x, y and z direction.
-) Sobel and Roberts edge detectors.
   The edge detection filters are applied analogous to the Laplace filtering method. The result may either replace the center voxel or it can be added to the old value, thereby providing edge enhancement.
-) Median filter.
   This filter provides strong suppression of random noise. The median grey value of a (3*3*3) area is assigned to the center voxel.
-) Minimum and Maximum filters.
   These algorithms are useful for automatic threshold operations. A local threshold, relative to local maximum and minimum voxel values can eliminate the effect of object illumination.

Apart from these local filters, the system also provides global operations. Global image processing operations include :

-) Histogram computation
   Each processor computes the histogram for the local data that is within the volume-of-interest and sends it back to the host, where the partial results are merged. The host can plot the result and use it as a base for the histogram equalization table. The histogram data will be transmitted to all nodes, where it can be used by image processing functions like automatic thresholding or edge detection.
-) Grey-scale transforms
   A conversion table is supplied to all processors to exchange the original voxel values against new ones. This operation may be used for example to filter out certain grey values or to provide a gamma correction. The conversion table may be edited manually or generated automatically from the histogram, providing the possibility of histogram equalization.

5 SYSTEM ARCHITECTURE

5.1 Introduction

This chapter gives a brief description of the modules in the system. Implementation details are given in the specific chapters dedicated to each module. Figure 5.1 shows the schematic representation of the voxel processor architecture. Ellipses are used to represent modules running in parallel. Parallelism was achieved in several ways, the most important step is dividing the object data into a number of (equally sized) sub-cubes, where each sub-cube has been assigned to one Transputer. Several processes may be running within each Transputer also, the reason for this is to make optimal use of the Transputers ability to do computations and perform communication at the same time. Apart from the sub-cube processors two more Transputers have been used to implement the Control and the Display module.

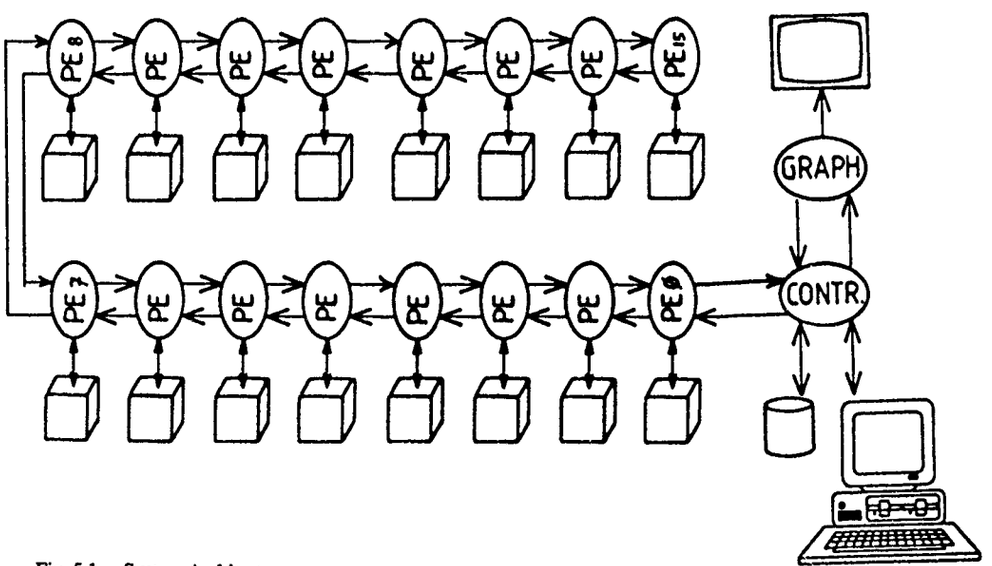


Fig. 5.1 System Architecture.

5.2 The Controller

The Controller (Fig. 5.2) is the user-interface between the host computer, a PC-AT, and the Transputer network that will perform the voxel visualisation. The Controller is in fact a 'software state machine'. Each state corresponds to a certain menu on the screen and a certain interpretation of the received user commands (keystrokes). The network is a full 'slave' that can only respond to commands from the Controller. Examples of possible Controller commands to the subcube processors are :

-) Store voxel data slice.
-) Render a view on voxel data set

Examples of possible Controller commands directed at the Graphics subsystem are :

-) Receive and display a slice of voxel data.
-) Receive and display a resulting image.
-) Receive and display text strings.
-) Receive and activate a new color table.

All results coming from the network must first pass the Controller before being displayed or stored on disk. Examples of possible results send to the Controller are :

-) Rendered image.
-) Computed histogram data.
-) Stored voxel data slices.
-) Error message.

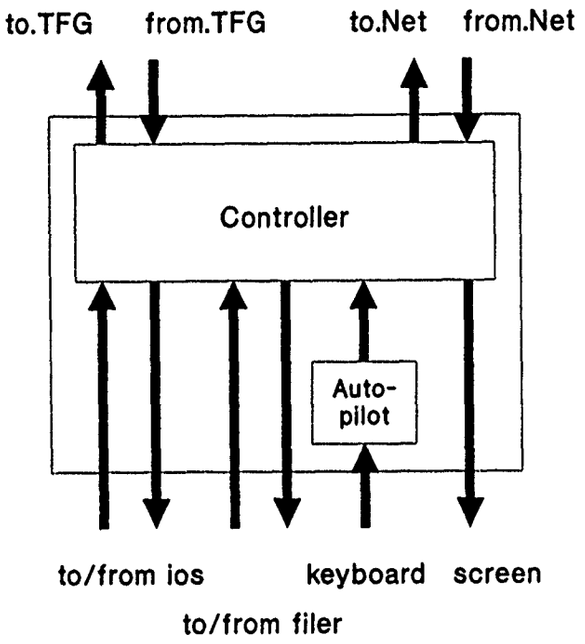


Fig. 5.2 Controller

5.3 The Subcube Processor

All commands to this Processing Element unit (PE) are exclusively send by the Controller. There are three main processes active inside a subcube module (Fig. 5.3), each of these processes is assigned to a specific function :

1) The Distributor, transfers commands and data across the network. All subcube processors are connected together in a tree or pipeline structure. This implies that each subcube processor must forward commands and data from one of its neighbours to its other neighbours. Except for global commands, like 'render an image', there are also commands meant for a specific node, like 'load a certain slice'. The Distributor can detect whether a command is global or intended for the local Transformation process and it will forward it accordingly. The Distributor is running in a continuous loop, ready to process incoming commands as soon as the previous command has been handled.

2) The Transformation process performs the object transformation, the 3D projection and the rendering. The module will completely generate the partial result for the assigned sub-cube and supply additional data that the mergers need to compute the final result. The sub-cube data (the voxels) are loaded only once for each new object and will not be changed during the transformations. The sub-cube Transputer will begin processing its data after receiving a command, which includes the transformed unit vectors and the selected type of rendering (e.g. front view, depth shade etc.). Beside performing the voxel-image transformation, this module is also used for the 3D image processing operations. For this end, memory is reserved to store both the original voxel-image and a processed version.
3) The Merger process receives 2D partial results from the local Transformation process and from its direct neighbour. These partial results will be merged into a new partial result which is transferred to the next Merger. When all partial results have been combined in this way, the last Merger will transfer the complete resulting image to the Controller process where it will be stored and displayed.

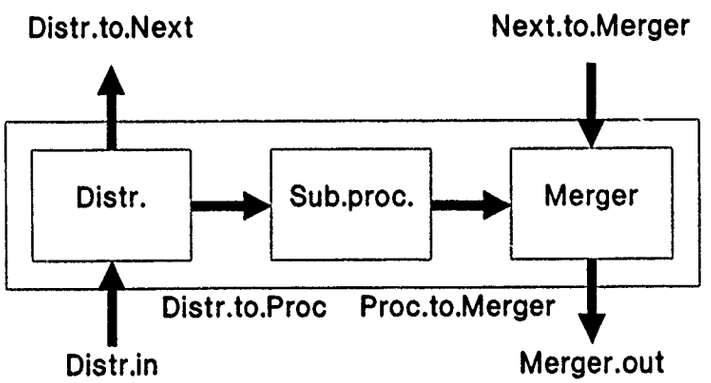


Fig. 5.3 The Subcube processor

5.4 The Graphics System

This unit (Fig. 5.4) is used for controlling a framebuffer in order to display the resulting images. A second function of this TFG board (Transputer Frame Grabber) is the acquisition of slice-data. The graphics board is in fact a framegrabber, capable of digitizing analogue video. Each recorded slice is the result of a number of (digitally) integrated video frames. This integration is used to reduce the noise level. The voxel data-set will be temporarily stored in the processor nodes, after which it may be viewed first before transferring the data to disk. The Graphics system is build as a 'server' that is continuously expecting input from the Controller. The input consists of a command tag, possibly followed by data. Examples of possible commands were given in the section on the Controller.

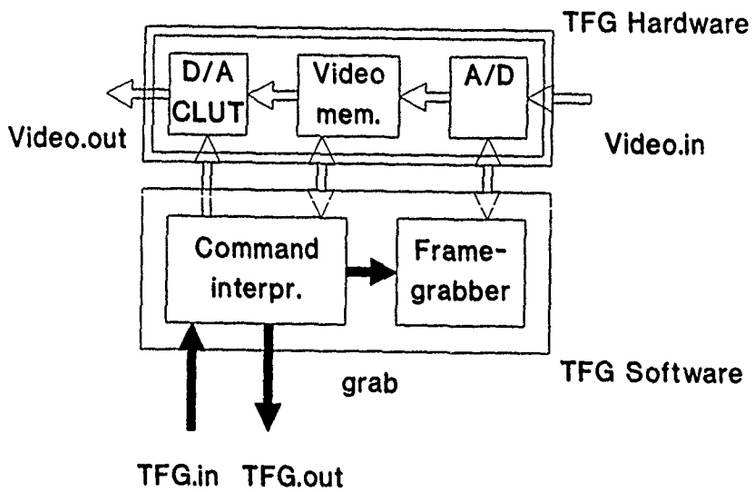


Fig. 5.4 Graphics system

5.5 Communication protocols

A complete list of all possible communication between the different modules is given below. The protocols are defined using identification tags and parameter lists. The tags have been defined as BYTE values in the libraries 'voxtag.tsr' (for the network) and 'tfgtag.tsr' (for the framegrabber). There are four types of communication defined which are identified by a 'main tag' :

-) command

   Commands have the following general structure :

   out ! command; sub.command[; parameters[; data]]

   Sub.commands indicate the type of command to be executed (e.g. pe.render) for which additional parameters and data are possibly supplied.

   In several cases a command will result in data being returned from the network (e.g. a rendered view) which must be received before continueing with a new command.

-) message

   The general structure is :

   out ! message; [8]BYTE ID.string;

   length::[length]BYTE string

   No response is expected for this communication. This communication type is used to transmit error- or debug- messages from a network processor to the operator display. The source of the message is given by the identifier 'ID.string'.

-) data

   Data communication has the following structure :

   out ! data; sub.data; [size]TYPE data

   No response is expected.

-) quit

   The 'quit' tag is the only component of this communication, it is echoed by the receiver to indicate that the receiving party has finished executing and will stop transmitting. This command will be send to all running processes in the network to guarantee an ordered shut-down of the whole system.

5.5.1 Controller to PE communication

The PE's expect only two main tags from the Controller :

-) commands
-) quit

Communication coming the PE's consists of :

-) messages
-) data
-) quit

The possible commands are separated into several groups :

-) Loading slice data into the network :
   -) to.net ! command; pe.load.slice; INT slice.nr;
      [SCDY][SCDX]BYTE slice

      response : none

      The 'INT slice.nr' indicates the number of the slice that will be send, its value is within [0..DZ>. Each PE node will use this number to decide if it has to load the slice locally or alternatively forward it to the next PE. Slice is an array of SCDY*SCDX bytes that represents the data of a slice. It is transmitted as SCDY arrays of SCDX bytes each, thus breaking up the large array in smaller portions to reduce intermediate storage requirements (speed is not a factor here, since this is limited by disk i/o anyhow).
-) Receiving slice data back from the network :
   -) to.net ! command; pe.send.slice; INT slice.nr

      response : data; slice.data;
      [SCDY][SCDX]BYTE slice

      This command is used to store the result of processed voxel data back to disk. Each PE that receives the command will either forward the command to the next PE or send the requested slice, if available.
-) Commands to manipulate stored voxel data sets :

   Two stored voxel data sets are present in the PE's, one is used as the 'source' and the other as the 'destination' for 3D image processing operations. All rendering operations use the 'destination' voxel data as input. Newly loaded voxel data set are stored as 'source'.
   -) to.net ! command; pe.reload.orig

      response : none

      The 'source' data is copied to the 'destination' array, thus enabling the visualisation of the 'source' data.

-) to.net ! command; pe.result.as.orig
response : none
The 'destination' data is copied to the 'source' array, thus enabling a new image processing operation on the present 'destination' data.

-) Commands to compute a view of the voxel data :
-) to.net ! command; pe.render; BYTE render.mode;
[3][3]REAL32 transform.vectors;
[3][2]INT transform.sizes;
INT z.cut.value; INT threshold.value
response : command; merge.image;
BYTE render.mode;
BOOL forward.merge;
[ID][ID]BYTE image
The transform.vectors array consists of the transformed unit vectors. The 'volume-of-interest' in the voxel data set is determined by start- and end values of voxel indices in x, y and z direction. These values are stored in the transform.sizes array. The remaining parameters (z.cut.value and threshold.value) are used by the rendering process to discard certain voxels. The response from the network has a number of additional parameters next to the actual data (the image). This is caused by the fact that the transmitted data is being processed by the 'mergers', which needed those parameters.

-) Commands to compute statistical data from the voxel data :
-) to.net ! command; pe.histogram;
[3][2]INT transform.sizes;
INT threshold.value
response : command; merge.histogram;
[256]INT histogram
Again, the response reflects the fact that the mergers are involved in the communication of the histogram data.

-) 3D image processing operations
The meaning of these operations is clearly indicated by their tags.
-) to.net ! command; pe.laplace
response : none
-) to.net ! command; pe.sobel
response : none

-) to.net ! command; pe.mean
response : none
-) to.net ! command; pe.median
response : none

-) Command to apply grey scale transforms to the voxel data set
-) to.net ! command; pe.grey.transform;
[3][2]INT transform.sizes;
[256]INT grey.table
response : none
All voxel data values (within the volume-of interest) are exchanged against the value that is found in the corresponding entry of 'grey.table'.

The 'quit' tag will close down the application :
-) to.net ! quit
response : quit

5.5.2 PE to PE communication :

PE's are connected together and exchange information using all defined main tags :

-) commands
All PE commands are results of Controller actions, no commands are transmitted on a PE's initiative. Most of the possible commands have been listed in the 'controller to PE interface', they are interpreted by the 'distributer' and forwarded to the next PE. The remaining commands are related to the merging of 'partial results'. These commands are transmitted and received by the 'merger' process. Ultimately such a command/data packet will arrive at the Controller.
-) merger.out ! command; merge.histogram;
[256]INT histogram.data
response : none
-) merger.out ! command; merge.image;
BYTE render.mode;
BOOL forward.merge;
[SCID][SCID]BYTE subcube.image
response : none

-) data

Data packets are exchanged between PE's to transmit stored data from one PE to the next and ultimately to the Controller. Possible data packets are :

-) out ! data; slice.data; [SCDY][SCDX]BYTE slice

response : none

-) messages

Any module within a PE may generate a message. Messages will be forwarded in the direction of the Controller by any receiving process within a PE.

-) quit

This tag must also originate from the Controller, it is forwarded from one PE to the next and it must also be echoed back to a sending PE.

5.5.3 Controller to TFG communication :

The TFG expects only two main tags from the Controller :

-) commands

-) quit

Communication from the TFG consists of :

-) messages

-) data

-) quit

The possible commands are separated into several groups :

-) Display a slice on the screen :

-) to.tfg ! command; tfg.load.slice;

[SCDY][SCDX]BYTE slice

response : none

The command is mainly used to display slices during the loading of a voxel data set from disk into the network.

-) Send a slice image from the screen :

-) to.tfg ! command; tfg.send.slice

response : data; slice.data;

[SCDY][SCDX]BYTE slice

When the TFG has grabbed a new image, this command is used to load the slice into the voxel data set.

-) Send an image from the screen :
   -) to.tfg ! command; tfg.send.image
      response : data; image.data; [ID][ID]BYTE image
      The image visible on the display is send, including text, grey scales etc. The image could be used for storage on disk.
-) Send an image to the screen :
   -) to.tfg ! command; tfg.load.image;
      INT start.y; INT nr.lines
      [nr.lines][ID]BYTE image
      response : none
      The transmitted image is displayed on the screen. The image is either read from disk or it may be the result of a rendering operation. The actual number of image lines is a variable ('nr.lines'), to prevent text or status lines visible in the display from being overwritten. The starting location in vertical direction for the first line is given by 'start.y'.
-) Initialise the screen :
   -) to.tfg ! command; tfg.mem.preset; INT color
      response : none
      The command initialises the display to the value 'color'. This command is mostly used to clear the screen (i.e. color set to 0).
-) Plot operations on the screen :
   -) to.tfg ! command; tfg.load.histogram;
      [256]INT table
      response : none
      This procedure is used to load and plot the contents of an array (e.g. a histogram or a color look-up table). The data is scaled to a maximum of 255 before plotting.
   -) to.tfg ! command; tfg.slice.border
      response : none
      When the TFG is used to grab a slice of data, it is helpful to draw a square on the screen to indicate the borders of this slice, since the slice is only a section of the screen. The dimension of the square is DY by DX pixels with its center corresponding to the center of the screen.

-) to.tfg ! command; tfg.load.string;
INT x.pos; INT y.pos;
INT background.color;
INT foreground.color;
length::[length]BYTE string

response : none

This command is used to display text on the screen. The x.pos and y.pos values indicate the upper right corner of the text. All characters have a size of 16*16 pixels.

-) to.tfg ! command; tfg.logo

response : none

The result is a TNO logo on the TFG screen.

-) to.tfg ! command; tfg.draw.line;
INT x.start; INT y.start;
INT x.end; INT y.end

response : none

A white line is drawn on the TFG screen connecting the given coordinates. The command is used to draw the reference coordinate system on the screen.

-) to.tfg ! command; tfg.grey.ref.bar

response : none

A reference grey scale is drawn in the lower left corner of the screen. This grey scale is useful to display the contents of the color look-up tables.

-) to.tfg ! command; tfg.test.pattern

response : none

The result is a test pattern on the TFG screen. The pattern has a grid to adjust the geometry of the monitor and grey- and color bars to adjust intensity and color.

-) Commands to control the frame grabber

-) to.tfg ! command; tfg.continuous.grab

response : none

The frame grabber switches to 'transparant' mode, this means the current video input signal is shown directly on the display.

-) to.tfg ! command; tfg.snapshot.grab

response : none

A snapshot (frozen image) is taken from the current video input signal. This image may now be used as slice data.

-) Commands to manipulate the color look-up tables.

The TFG stores four local color look-up tables. At any time only one table is active in the conversion of greyvalues to pseudo-colors.

-) to.tfg ! command; tfg.select.lut;

INT active.lut.nr

response : none

The valid range for active lut.nr is [0..3], this value selects one of the stored tables. The table will be used immediately to convert the data in the frame memory (BYTES) to RGB values on the screen.

-) to.tfg ! command; tfg.load.lut;

INT lut.nr;

[3][256]INT color tables

response : none

This command is used to load new data in one of the local color tables of the TFG. The lut.nr values indicates which table is selected, the three arrays that follow it represent the table contents for red, green and blue. Table entries are limited to the [0..63> range (i.e. 6 bits of RGB resolution).

The application will finish execution after receiving a 'quit' tag :

-) to.tfg ! quit

response : quit

5.6 Configuration files

The system is very flexible in the dimensions of the objects that are to be transformed. The voxel object dimensions are an important factor for the software, since many internal data structures are dependant of them. Only four dimensions must be provided within the configuration file 'vox_cnst.tsr' to select a different object size. These values are :

-) The dimensions of the voxel object

-) The size of the resulting image

Several other values are automatically derived from these values. This is illustrated in listing 5.1. In the current version, a change of these library values does require a recompilation. It is possible to adapt the software to provide a run-time selectable object size.

Some restrictions must be considered for the object sizes :

-) The total amount of available memory for a Subcube node must not be exceeded (currently 2 MByte).
-) The projection of the transformed object must fit within the size of the resulting image, since no clipping has been provided sofar.
-) Some optimisations have been implemented in the current version of the voxel-processor that require the object dimensions to be a power of two. This can be changed easily without a significant performance penalty.

Listing :5.1

List of File : voxcnst.tsr

File Last Modified : 22-11-90

```
{{{  VAL's  for voxel object  (128*128*128)
-- Object dimensions in x, y and z                                    :
VAL         DX                    IS                          128    :
VAL         DY                    IS                          128    :
VAL         DZ                    IS                          128    :
--          Resulting image dimension                                :
VAL         ID                    IS                          256    :

-- All remaining system constants are derived from the values given above    :

-- Object dimension                                                  :
VAL         DX1                   IS                        (DX-1)   :
VAL         DY1                   IS                        (DY-1)   :
VAL         DZ1                   IS                        (DZ-1)   :
-- Number of voxels in object                                        :
VAL         VOL                   IS                ((DX * DY) * DZ)   :
-- Object center                                                     :
VAL         H.DX                  IS                        (DX/2)   :
VAL         H.DY                  IS                        (DY/2)   :
VAL         H.DZ                  IS                        (DZ/2)   :
-- Resulting image                                                   :
VAL         ID1                   IS
--          Result image center                                      :
VAL         H.ID                  IS                         (ID/2)  :
VAL         H.ID.REAL             IS              (REAL32 ROUND H.ID)  :
-- Subcube object dimensions in x, y and z                           :
VAL         SCDX                  IS                           DX    :
VAL         SCDY                  IS                           DY    :
VAL         SCDZ                  IS            (DZ / NR.OF.NODES)   :
VAL         SCDX1                 IS                      (SCDX-1)   :
VAL         SCDY1                 IS                      (SCDY-1)   :
VAL         SCDZ1                 IS                      (SCDZ-1)   :
-- Number of voxels in Subcube object                                :
```

```
VAL         SCVOL                 IS              ((SCDX * SCDY) * SCDZ)    :
-- Subcube result image dimensions                                          :
VAL         SCID                  IS                                  ID    :

-- Number of bytes in a sector on disk                                      :
VAL         SECSIZ                IS                                 512    :
VAL         SECT.IN.SC.PLANE      IS             ((SCDX * SCDY) / SECSIZ)    :
VAL         NO.OF.SECTS.IN.SC     IS                    (SCVOL / SECSIZ)    :
VAL         NO.OF.SECTS.IN.OBJ    IS                    (  VOL / SECSIZ)    :
:::
```

Changing the number of subcube processors is easy, because of the modular set-up. This results in a flexible cost/performance ratio. The software is identical for all Transputers, parameters are used to compute which actual slices are to be stored and processed on a specific node. Several parameters are used to control the configuration of the system. These parameters are stored in the library 'vox_conf.tsr' (Listing 5.2), they control :

-) The number of subcube processors used in the system

-) The processor topology

Changing these parameters does require a recompilation of the system software (since the data is distibuted differently) and of the network configuration code ('PROGRAM vox_net.tsr'). The table 'LAST' is used to enable an easier transfer from a pipeline to a tree topology. The BOOL's found in the table indicate that a processor node is at the end of a pipe of processors and should not attemp to forward any messages.

The restrictions in the number of nodes are :

-) The total amount of available memory for a Subcube node must not be exceeded (currently 2 MByte).

-) There must is a minimum of one 'slice' per node, this implies that a maximum of 'DZ' subcube nodes can be used for any object.

Listing : 5.2

List of File : voxconf.tsr

File Last Modified : 22-11-90

```
{{{  VAL's  for 16 processor configuration
-- Total number of subcube processors                                    :
VAL   INT NR.OF.NODES           IS                                  16    :
-- Table to indicate that subcube processors is at the end of the processor
   pipeline                                                              :
VAL[NR.OF.NODES]BOOL     LAST.ARRAY          IS
         [  FALSE,       FALSE,              FALSE,              FALSE,
            FALSE,       FALSE,              FALSE,              FALSE,
            FALSE,       FALSE,              FALSE,              FALSE,
            FALSE,       FALSE,              FALSE,               TRUE   ] :
}}}
```

5.7 Implementation Remarks

-) The Voxel processor is built up entirely with of-the-shelf hardware. Each MTM-2 processor board [12] offers two T800's with 2 MByte of memory each, so two PE's are loacted on the board. Other boards used, are the Display System TFG [13] with an on-board framegrabber, a T800 and 1 MByte of video-ram and a TPM-4 processor board [14] with 4 MByte memory for the controller. Physically the system consists of a 19" cabinet with 10 single euro-sized boards installed. The host system in the Voxel-processor is an IBM-AT.

-) New or improved rendering operations can be added easily, since the system has been set up very modular. The main restriction is the fact that, in the current implementation, a sub-cube processor has no direct access to (voxel) data located in other PE's.

-) A communication layer is integrated into the system. This layer provides data and command transport to all processes, and it is also capable of sending (debug) messages from any process to the operator screen.

6 SOFTWARE DESCRIPTION

6.1 Introduction

The Voxelprocessor software is written in OCCAM 2 [7], developed under the MULTITOOL 5.0 system running on a PC-AT. This development system uses a 'Folding Editor' [15] to write the code. The folding editor is in our opinion a very useful tool during development and presents a clear overview of the code on the console screen. The main advantage of a folding editor is that it supports a 'top-down' view of the software. At each level, the user only sees the main structure of the program, with details folded away. The contents of a fold can be described in the fold header. A closed fold is represented by "... Closed Fold Header". A fold lister program is available for documentation purposes. This tool generates listings of the source code with folds opened or closed under user control. These listings will be used here to describe the basic operation of the software, without going into every detail. The top-down approach of the folding editor will be used here for the description of the software : the general structure of the programs will be explained first, with details temporarily hidden in folds. Only the most important parts will be explained, different cases of the same basic operation will not be treated each time over.

The top level of the Voxel-processor software is shown in listing 6.1, it consists of three parts :

1) The Libraries.
   Constants, variables or procedures that are (or could be) used in several processes of this application were developed as Library code. Examples are message communication tags, that must be known both to the transmitting- and to the receiving process. The configuration details are also stored in a Library, for example : the size of the voxel-dataset and the number of processors in the network.
2) The EXE code
   This part of the software runs on the 'root' Transputer, that is the processor with access to the Host PC-AT computer. The EXE runs under control of the Transputer Development System (TDS), also known as 'MULTITOOL 5.0'. The EXE code is used to control the operation of the Voxel-processor.

3) The PROGRAM code.

This is the code that runs on the rest of the Transputer network. It consists of three parts :

-) The code for the SubCube processors.

-) The code for the Framegrabber/Display processor.

-) The Network configuration description.

Listing : 6.1

List of File : "voxel.tsr"

File Last Modified : 22-11-90

```
---  Libraries :
...F voxconf.tsr    --- number of processors in the network
...F voxcnst.tsr    --- dimensions of the Voxel dataset
...F voxproc.tsr    --- rendering procedures for Network processors
...F tfgtag.tsr     --- communication tags Controller <-> Framegrabber
...F voxtag.tsr     --- communication tags Controller <-> Network
...F menuio.tsr     --- menu display and parameter i/o procedures
...F dewrite.tsr    --- procedures to display debug messages

---  System :
...  EXE vox_cntr.tsr     --- code running on the root processor
...  PROGRAM vox_net.tsr  --- code running on the network processors
```

6.2 The EXE Controller

The EXE Controller is the origin of all commands to the system and it is the destination of all computed results. The operator communicates with the 'user-interface' part, which interprets and executes his commands. The user-interface consists of a menu system, that shows all possible commands and the presently active parameter settings. The operator may move through the menu options by pressing function keys, arrow keys or characters. The selected menu entry is indicated by a high-lighted bar. Operations can be started by pressing the 'return' key on an menu entry. The Controller is the only process that has access to the keyboard, screen and disks of the host computer (the PC-AT). After loading the sub-cube processors with voxeldata from the disk or via the framegrabber, the object may be rotated and viewed interactively under the command of the Controller. The resulting images are received by the Controller and send to the Framebuffer board for display. The results can also be stored on disk, and read in again at a later time.

Listing : 6.2

List of File : "vox_cntr.tsr"

File Last Modified : 22-11-90

```
...  link channel numbers
{{{  channels
CHAN OF ANY from.auto.pilot :

CHAN OF ANY from.tfg, to.tfg :
PLACE to.tfg    AT  linkout1 :
PLACE from.tfg  AT   linkin1 :

CHAN OF ANY from.net, to.net :
PLACE to.net    AT  linkout2 :
PLACE from.net  AT   linkin2 :
}}}

...  SC T8 auto.pilot
...  SC T8 control.tsr        --- user interface (tfg)
--  manual & autopilot control
PAR
  auto.pilot (keyboard, from.auto.pilot)
  controller (from.auto.pilot, screen,
              from.filer, to.filer, from.ios, to.ios,
              from.net, to.net, from.tfg, to.tfg)
```

Listing 6.2 shows the toplevel structure of the EXE Controller. The function of the channels to the host (keyboard, screen, to/from.filer and to/from.ios should be clear, to/from.net are used for sending commands and data to the Transputer network. To/from.tfg are used for sending images to the Framebuffer for display and to receive images that were grabbed from the camera. In this implementation, an Autopilot has been included. Its function will be explained later. If desired, a Controller EXE without the Autopilot could be build by simply removing the SC auto.pilot and replacing the from.auto.pilot channel with the keyboard channel.

6.2.1 The SC Controller

The basic structure of the SC Controller is shown in listing 6.3.

Listing : 6.3

List of File : "control.tsr"

File Last Modified : 22-11-90

```
PROC controller ( CHAN OF INT keyboard,
                  CHAN OF ANY screen,
                  from.filer, to.filer,
                  from.ios, to.ios,
                  from.net, to.net, from.tfg, to.tfg)

  ...  header.tsr          -- Declarations
  ...  menu.tsr            -- Menu related stuff
  ...  net.tsr             -- Receive data/mess from network
  ...  PROC rotate routine -- Subcube transform parameters
  ...  PROC init.unit.vect
  ...  PROC init.rotate.param
  ...  text.tsr            -- Text markers
  ...  displ.tsr           -- Display parameters on screen
  ...  lutio.tsr           -- Init/Load/Save LUT tables
  ...  imagio.tsr          -- Load/Save images
  ...  objectio.tsr        -- Load/Save Object
  SEQ
    ...  init variables
    ...  show banner
    running := TRUE
    WHILE running
      ...  declarations
      SEQ
        menu ()           --- draw menu's
        PRI ALT
          from.net ? tag
            receive.mess (tag)  --- action on network message
          keyboard ? ch
            ... F key.tsr       --- action on keyboard command
:
```

The Controller starts with some initialisations of variables, it then sends a banner to the PC display and shows a Logo on the Framegrabber screen. From that moment on, the Controller runs in a loop, displaying the menu structure ('menu' procedure) and awaiting user commands to be executed ("... key.tsr" fold). Alternatively the Controller can also receive messages from the network ("... net.tsr" fold). The messages could either be results from an ongoing computation or possibly error messages. The Controller consists of several basic building blocks :

-) The menu system.

"... menu.tsr" fold, (6.2.1.1).

-) The network communication part.

"... net.tsr" fold, (6.2.1.2).

-) The action on keyboard input fold.

"... key.tsr" fold, (6.2.1.3).

The disk interface routines, consisting of :

-) Voxel dataset load and store.

"... objectio.tsr" fold, (6.2.1.4).

-) Result images load and store.

"... imageio.tsr" fold, (6.2.1.5).

-) Look-up table load and store.

"... lutio.tsr" fold, (6.2.1.6).

-) Rotated unit vectors calculation

"... rotate.tsr" fold, (6.2.1.7).

-) Parameter display on Framegrabber screen.

"... display.tsr" fold, (6.2.1.8).

The detailed operation of the Controller will be described through these building blocks.

6.2.1.1 Menu system

The menu system is the backbone of the Controller, all operations are guided through it. The menu system is in fact a 'software state machine'. Its state is represented by the variable 'state'. Each state corresponds to a certain menu on the screen (displayed by the 'menu ()' procedure) and a certain interpretation of the received user commands (performed by the "... key.tsr" fold). All possible states have been given names, corresponding to an integer value.
These names and values are :

```
{{{  state names
VAL MAIN.state       IS    1 :
VAL RENDER.state     IS    2 :
VAL OPTIONS.state    IS    3 :
VAL COLOR.state      IS    4 :
VAL GRABBER.state    IS    5 :
VAL OBJECT.state     IS    6 :
VAL INTER.ROT.state  IS   21 :
VAL INTER.CUT.state  IS   22 :
VAL INTER.SLC.state  IS   23 :
VAL EDIT.state       IS   45 :
VAL MAN.GR.state     IS   51 :
VAL AUTO.GR.state    IS   52 :
VAL EDGE.state       IS   60 :
VAL NOISE.state      IS   61 :
VAL GREY.SCL.state   IS   64 :
}}}
```

Each state also has a menu list assigned to it. This list will be displayed on the operator screen when the state becomes active. The entries in the list show the possible user commands that can be selected in this state. 'MENU.TABLE' is used to declare all entries (i.e. text) of all the menu's. The menu's must have a standard size of max. 10 entries per menu, and a max. of 30 characters per entry. This entry data is stored in the 'menu.table' variable during run time, so that it's contents may be changed (when parameter values in the menu's are changed).

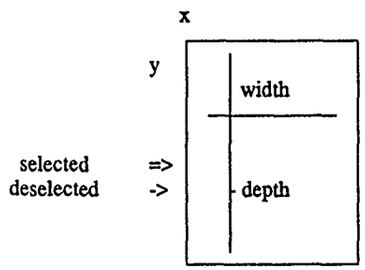
The contents of the MENU.TABLE is partly shown here :

```
VAL MENU.TABLE  IS {{{   main
                   [["   Load Object                ",
                     "   Save Object                ",
                     "   Load Image                 ",
                     "   Save Result                ",
                     "   Dos Shell                  ",
                     "   Quit                       ",
                     "                              ",
                     "                              ",
                     "                              ",
                     "                              " ],
                    }}}
                    ...  rotate
                    ...  options
                    ...  color
                    ...  grabber
                    ...  object
                    ...
                     other menu's
                    ...
                     ] :
```

The 'MENU.DATA' constant array is used to store several parameters of each menu list :

-) The size of the menu, the number of entries (depth) and its width.
-) The upper left corner position of the menu border when it is displayed on the screen.
-) The selected item in the menu and the previously selected item, used to show the selected entry in inverse video.

The meaning of the parameters is clearly illustrated in the figure below :



For MAIN.state, the values are as follows :

```
MENU.DATA [MAIN.state]    IS [ 0, 3, 17, 6, 0, 1]
```

The fields of the 'MENU.DATA' and 'menu.data' array are accessed using the following definitions :

```
VAL MENU.X          IS   0 :
VAL MENU.Y          IS   1 :
VAL MENU.WIDTH      IS   2 :
VAL MENU.DEPTH      IS   3 :
VAL MENU.SELECT     IS   4 :
VAL MENU.DESELECT IS   5 :
```

The 'menu.data' variable array holds the menu parameter data during run time, so that it may be changed.

The menu structure on the screen can be seen as a 'tree' of menu's, since some menu entries will result in new (sub-) menu's being opened on selection. The presently active menu tree is stored in the 'menu.tree' variable array as a sequence of state numbers. The menu tree is used each time the menu structure has to be redrawn. The present depth of the tree is stored in 'menu.tree.depth'. This value is changed when a (sub-) window is added to or removed from the tree.

Whenever the menu tree is redrawn on the screen, this can be done in tree defined modes : REDRAW, ADD or UPDATE. The different modes have been implemented to improve the speed of the redraw action if possible. The differences are :

-) REDRAW
   A complete redraw of the menu tree, starting from an empty screen. This mode is chosen when a 'function' or 'arrow' key is used to switch between menu's.
-) ADD
   This mode is selected to add a new (sub-) window to the menu screen. The windows that are present on the screen are overwritten if they are overlapped by the newly added one.
-) UPDATE
   This mode only changes the selection pointer in the presently active menu. It is probably the most often used redrawing mode.

The menu redraw mode is passed to 'PROC menu ()' via the 'menu.adapt' variable. This variable is set to one of the above options depending on the last keyboard action.
Listing 6.4 shows how 'PROC menu ()' responds when the REDRAW option is active, as mentioned, the other two modes will perform only parts of this complete redraw.

Listing : 6.4

```
{{{  PROC menu                        --- redraw menu (different modes)
PROC menu ()
  {{{  PROC top.level ()
  PROC top.level ()
    ---  draw main menu names at top of the screen
    ...  VAL top.line
    ...  VAL bottom.line
    SEQ
      screen ! tt.goto; 0; 0
      write.full.string(screen, top.line)
      {{{  top level menu names
      write.full.string(screen, "||   MAIN  || RENDER |
                                | OPTIONS || COLOR  |
                                | GRABBER || OBJECT ||*c*n")
      }}}
      write.full.string(screen, bottom.line)
  :
  }}}
  IF
    menu.adapt = REDRAW
      {{{  redraw complete menu tree
      SEQ
        top.level ()
        screen ! tt.clear.eos         --- clear the rest of the screen
        --- show all menu's in the menu tree
        SEQ menu.count=0 FOR menu.tree.depth
          {{{  show the menu
          INT X, Y :
          menu.active   IS menu.tree [menu.count]    :
          active.data   IS menu.data [menu.active]   :
          menu.select   IS active.data [MENU.SELECT] :
          menu.width    IS active.data [MENU.WIDTH]  :
          menu.depth    IS active.data [MENU.DEPTH]  :
          SEQ
            --- goto upper left corner of the menu
            X := active.data [MENU.X]
            Y := active.data [MENU.Y]
            screen ! tt.goto; X; Y
            {{{  display top menu border
            write.full.string(screen,"*#C9")     --- "╔"
```

```
            write.full.string(screen,
                 [ DOUBLE.LINE FROM 0 FOR menu.width ] )      --- "═══"
            write.full.string(screen,"*#BB")     --- "╗"
            Y := Y + 1
            }}}
            --- show all entries of the menu
            SEQ entry=0 FOR menu.depth
              {{{  left border, menu entry, right border
              SEQ
                screen ! tt.goto; X; Y
                write.full.string(screen,"*#BA")  --- ║
                ---  entry selection
                IF
                  entry = menu.select
                    ...  write menu entry in inverse video
                  TRUE
                    ...  write menu entry normal
                write.full.string(screen,"*#BA")  --- "║"
                Y := Y + 1
              }}}
            ...  bottom menu border    --- "╚═ = = ═╝"
          }}}
        }}}
      menu.adapt = ADD
        ...  add new menu : menu.tree [menu.tree.depth-1]
      menu.adapt = UPDATE
        ...  update select pointer menu : menu.tree [menu.tree.depth-1]
:
```

6.2.1.2 Network communication

Listing 6.5 shows the contents of the 'net.tsr' file. This file consists of two procedures that the Controller uses to receive data or messages from the network. The 'PROC receive.mess ()' procedure runs in the main loop of the Controller and is ready to receive messages at any time. These messages will only occur when the system is in debug mode and the network processors send error- or progress messages to the screen. When a message is received, it causes the display of a special window at the bottom of the screen. The message has a standard protocol consisting of a source identification string (ID.String) and a text string containing the actual information. The ID.string is displayed in the window border, while the message itself is written inside the window. The procedure waits for a keystroke to give the operator time to read the message, and continues normal execution after removing the message window.

The second procedure, 'PROC receive ()', is activated by the Controller when it expects certain data from the network in response to given commands. The procedure will check the actual received data type against the expected type. If the "received.type" is equal to the "expected.type" then the variable "ok" is set "TRUE". Any expected data may be preceded by one or more (unexpected) messages. Error messages are displayed whenever unexpected or unknown tags are received. The listing shows only the main structure of the procedure since it is basically identical for all possible types. The protocols for the received network data have been described in chapter 5.5.

Listing : 6.5

List of File : "net.tsr --- Receive data/mess from network"

File Last Modified : 4-11-1990

```
{{{  VAL  expected types for PROC receive
--- These are the possible data types that the
--- controller can expect to receive from the network.
VAL receive.message   IS   0 (BYTE) :
VAL receive.merge     IS   1 (BYTE) :
VAL receive.image     IS   2 (BYTE) :
VAL receive.slice     IS   3 (BYTE) :
VAL receive.histogram IS   4 (BYTE) :
VAL receive.quit      IS   5 (BYTE) :
}}}

{{{  PROC receive.mess ()
PROC receive.mess (BYTE tag)
  --- This unit will only receive messages from the network.
  --- Any other received data type will cause an error report.
  IF
    tag = message
      {{{  process one message
      {{{  VAR's
      INT any, x, y :
      INT length :
      [8]BYTE ID.string :
      [80]BYTE string :
      }}}
      SEQ
        -- short.beep()
        {{{  draw message window
         screen ! tt.goto; 0; 19; tt.clear.eos
        ...  top border     --- "┌── MESSAGE FROM       ──┐
        ...  middle         --- "│                       │
        ...  bottom         --- "└── Strike Any Key... ──┘
```

```
            }}}
            {{{  show sender id
            from.net ? ID.string
            screen ! tt.goto; 42; 19  -- correct position in window border
            write.full.string (screen, ID.string)
            }}}
            {{{  init x, y
            x := 2
            y := 20
            screen ! tt.goto; x; y
            }}}
            {{{  receive and show message string
            from.net ? length::string
            write.full.string(screen, [string FROM 0 FOR length])
            }}}
            {{{  wait for any key to continue
            keyboard ? any
            screen ! tt.goto; 0; 19; tt.clear.eos
            }}}
          }}}
        tag = command
          {{{  error
          SEQ
            screen ! tt.goto; 0; 0; tt.clear.eos
            write.full.string(screen,
              "ERROR - Message expected, Command tag from network")
            wait.any (keyboard, screen)
          }}}
        tag = quit
          ...  error
        TRUE
          ...  error
:
}}}


{{{  PROC receive ()
PROC receive (CHAN OF ANY to.contr, VAL BYTE expected.type, BOOL ok)
  --- This unit will receive messages, or data from the network.
  --- It will check the actual received data type against the expected
  --- type. If (received.type = expected.type) then ok := TRUE.
  --- Any expected data may be preceded by one or more messages.
  BYTE tag :
  SEQ
    ok := FALSE
    to.contr ? tag  --- receive
    {{{  process possible preceding messages
    WHILE (tag = message) AND (NOT (expected.type = receive.message))
      SEQ
        ...  process one (unexpected) message
        to.contr ? tag   --- receive more
```

```
}}}
{{{  process expected data
IF
  tag = message
    {{{  process message
    IF
      (expected.type = receive.message)
        ...  ok, receive message
      TRUE
        {{{  error   --- this can not happen !!
        SEQ
          screen ! tt.goto; 0; 0; tt.clear.eos
          write.full.string (screen,
                   "ERROR - Unexpected message tag from network ")
          wait.any (keyboard, screen)
        }}}
    }}}
  tag = command
    {{{  process command
    SEQ
      to.contr ? tag
      IF
        tag = merge.image
          {{{  receive merged image
          IF
            (expected.type = receive.merge)
              ...  ok, receive image
            TRUE
              ...  "ERROR - Unexpected sub.tag from network "
          }}}
        tag = merge.histogram
          {{{  receive histogram
          IF
            (expected.type = receive.histogram)
              ...  ok, receive histogram
            TRUE
              ...  "ERROR - Unexpected sub.tag from network"
          }}}
        TRUE
          ...  "ERROR - Unknown command sub.tag from network"
    }}}
  tag = data
    {{{  process data
    SEQ
      to.contr ? tag
      IF
        tag = image.data
          {{{  receive image
          IF
            (expected.type = receive.image)
              ...  ok, receive image
```

```
                TRUE
                  ...  "ERROR - Unexpected sub.tag from network"
              }}}
            tag = slice.data
              {{{  receive slice
              IF
                (expected.type = receive.slice)
                  ...  ok, receive slice
                TRUE
                  ... "ERROR - Unexpected sub.tag from network"
              }}}
            TRUE
              ... "ERROR - Unknown data sub.tag from network"
        }}}
      tag = quit
        {{{  process quit
        IF
          (expected.type = receive.quit)
            ...  ok, quit
          TRUE
            ... "ERROR - Unexpected Quit tag from network"
        }}}
      TRUE
        ...  "ERROR - Unknown main tag from network"
  }}}
:
}}}
```

6.2.1.3 Keyboard action

Listing 6.6 shows the '... key.tsr' fold. Incoming characters are checked in a CASE structure. Function keys (F2...F6) cause a direct change of state. With these keys the user may switch from any (sub) menu to any other menu. The exact operation can be found in the fold '... ch = Function key' (Listing 6.6.1), with the first case (ch = F2) written out. If the received character is not a function key, the CASE construct in '... key.tsr' will fall through to the '... ch = State specific key'. Listing 6.7 shows the contents of this fold.

Listing : 6.6

List of File : 'key.tsr'

File Last Modified : 22-11-90

```
{{{  VAL declarations
...  Function keys
...  Other keys
...  Other values
}}}
IF
  ...  ch = Function key
  ...  ch = State specific key
```

The result of pressing the F2 key is a switch to 'state = MAIN.state' and the according settings of certain variables for the correct update of the menu structure on the screen. A special key (#) has been reserved to bring the Controller in a debug mode, where intermediate results will be displayed on the user screen.

Listing: 6.6.1

```
List of Fold : 'ch = Function key'

ch = F2
  {{{  MAIN.state
  SEQ
    state := MAIN.state
    menu.tree[0] := MAIN.menu
    menu.tree.depth := 1
    menu.adapt   := REDRAW
  }}}
ch = F3
  ...  RENDER.state
ch = F4
  ...  OPTIONS.state
ch = F5
  ...  COLOR.state
ch = F6
  ...  GRABBER.state
ch = F7
  ...  OBJECT.state
ch = DEBUG.MODE
  ...  debug.mode On/Off
```

The '... ch = State specific key' fold starts with some abbreviations to the active menu : the number of entries, the presently high-lighted entry etc. This is useful, to enable quick access to the data.

The process then jumps to a unique branch for each possible state of the menu state machine. The received characters will be interpreted differently depending on the currently active state.

Listing : 6.7

List of Fold : "ch = State specific key"

```
TRUE
  {{{  menu abbreviations
  menu.active    IS menu.tree [menu.tree.depth - 1] :
  active.data    IS menu.data [menu.active]     :
  menu.depth     IS active.data [MENU.DEPTH]    :
  menu.select    IS active.data [MENU.SELECT]   :
  menu.deselect  IS active.data [MENU.DESELECT] :
  }}}
  {{{F state case
  IF
    state = MAIN.state
      ...  interpret keys and execute actions
    state = RENDER.state
      ...
    state = OPTIONS.state
      ...
    state = COLOR.state
      ...
    state = GRABBER.state
      ...
    state = OBJECT.state
      ...
    state = INTER.ROT.state
      ...
    state = INTER.CUT.state
      ...
    state = INTER.SLC.state
      ...
    state = EDIT.state
      ...
    state = MAN.GR.state
      ...
    state = GREY.SCL.state
      ...
    state = EDGE.state
      ...
    state = NOISE.state
      ...
    TRUE
      ... error
  }}}
```

The '... interpret keys and execute actions' branch is selected when the 'MAIN.state' is active. Listing 6.8 shows the contents of this fold. As described earlier, the received characters will be interpreted according to the possible commands in the 'MENU.state'. The basic structure of this fold is not only used for 'MAIN.state', but for all other states also. The other states will therefore not be described explicitly. The possible keys for each state can be ordered in a number of categories :

1) Arrow Left or Right : switch to the neighbour state as indicated on the screen layout. The result is identical to pressing the corresponding function key.
2) Arrow Up or Down : move the high-lighted bar to the previous/next menu entry. The high-light moves from top to bottom at the last or first entry, if necessary. Using the arrows keys does not activate the selected entry.
3) Character keys are used to get a direct and faster selection of an entry. This is an alternative to using the up/down arrow. The first character of an entry is normaly used for this selection.
4) Pressing the 'return' key activates the high-lighted menu entry.
5) Keys that are not trapped in the CASE structure, will be considered 'illegal' and cause a 'beep' on the console.

Listing 6.8

```
{{{  interpret keys and execute actions
IF
  {{{  switch menu
  ch = ft.left
    {{{
    SEQ
      state := OBJECT.state
      menu.active  := OBJECT.menu
      menu.adapt   := REDRAW
    }}}
  ch = ft.right
    {{{
    SEQ
      state := RENDER.state
      menu.active  := RENDER.menu
      menu.adapt   := REDRAW
    }}}
  }}}
  {{{  switch selected item
  ch = ft.up
    {{{
    SEQ
```

```
    menu.deselect := menu.select
    menu.select   := menu.select - 1
    IF
      menu.select = MINUS.ONE
        menu.select := menu.depth - 1
      TRUE
        SKIP
    menu.adapt    := UPDATE
  }}}
ch = ft.down
  {{{
  SEQ
    menu.deselect := menu.select
    menu.select   := (menu.select + 1) \ menu.depth
    menu.adapt    := UPDATE
  }}}
(ch = KEY.L) OR  (ch = KEY.l)
  {{{ Load Object high-lighted
  SEQ
    menu.deselect := menu.select
    menu.select   := 0
    menu.adapt    := UPDATE
  }}}
(ch = KEY.S) OR  (ch = KEY.s)
  ... Save Object high-lighted
(ch = KEY.I) OR  (ch = KEY.i)
  ... Load Image high-lighted
(ch = KEY.R) OR  (ch = KEY.r)
  ... Save Result high-lighted
(ch = KEY.D) OR  (ch = KEY.d)
  ... DOS Shell high-lighted
(ch = KEY.Q) OR  (ch = KEY.q)
  ... Quit high-lighted
}}}
{{{  select item
(ch = return) AND (menu.select = 0)
  ...  Load Object action
(ch = return) AND (menu.select = 1)
  ...  Save Object action
(ch = return) AND (menu.select = 2)
  ...  Load Image action
(ch = return) AND (menu.select = 3)
  ...  Save Result action
(ch = return) AND (menu.select = 4)
  ...  DOS Shell action
(ch = return) AND (menu.select = 5)
  ...  Quit action
}}}

{{{  ILLEGAL CHOICE
TRUE
```

```
    SEQ
      menu.adapt    := UPDATE
      short.beep (screen)
  }}}
}}}
```

The result of pressing the 'return' key on the 'Load Object' menu entry is described in listing 6.9. All actions requiring user supplied parameters, like filenames, angles or colour tables, will open a special window at the bottom of the screen. The size and position of this window is controlled by the constants X, Y, WIDTH and HEIGHT, which are defined at the start of the '... Load Object action' fold. These constants might be different for other actions, depending on the number of expected user parameters. The procedure 'draw.window ()' will perform the actual drawing of the window borders, using the already mentioned constants. Several procedures have been defined to perform user i/o of parameters within the borders of the window.

These procedures are :

-) get.filename.
   This procedure will take a given prompt, display it and wait for a user supplied file name. Depending on the type of file i/o (read or write), it will check for the existence of the requested file or alternatively create a new file with this name. The procedure will display error messages within the window borders if necessary, and exit with the 'Error' variable set to a non-zero value after a certain number of consecutive errors. If all goes well, the user supplied filename will be assigned to the string variable 'File.name' and its length will be found in 'File.name.len'
-) get.string
   This procedure is identical in function to the previous one, but it will just read in an unchequed string.
-) get.int
   This procedure is identical in function to the first one, with the difference that it is meant to read integer values. The integer values will be chequed against a minimum and maximum value that is supplied as a procedure parameter.
-) get.bool
   This procedure is again identical to 'get.integer', but intended to read in Boolean values ('Y', 'y', 'N' or 'n').

The screen area occupied by the window can be erased again by the 'clear.window' procedure, which uses the same parameters as 'draw.window'. A faster method however is to directly access the 'screen' channel and clear the area below the cursor with :

screen ! tt.goto; X; Y; tt.clear.eos

Two procedures have been added to activate and disable the cursor on the PC screen, these are 'cursor.on' and 'cursor.off'. During normal menu control, the cursor will be turned off, the current position on the screen is indicated by the high-lighted bar. For user i/o however the cursor should be visible in the parameter window again. The two procedures use the 'to/from.ios' channels, which provide direct access to the PC's interrupt mechanism. Through 'INT 10' it becomes possible to change the cursor size to zero, thus making it invisible. The window and parameter input procedures are located in the '... LIB menu.io' fold.

After entering a valid file name (Error = 0), the '... Load Object action' fold will continue by reading the requested voxel dataset from disk and loading it in the Transputer network. As mentioned before, the data will be read in on a slice by slice basis, displaying each slice on the framegrabber screen and sending it to the correct Processor Element (PE). This combination of actions is performed by the 'load.object' procedure (6.2.1.4). After successfully loading a new object (Error = 0), several variables will need to be reinitialised and the operator will be requested to supply a name for the voxel dataset. The initialisation procedures will be described in following paragraphs. The user supplied name is embedded in a header string that is continuously displayed at the top of the framegrabber screen. This header string is build by 'create.top.line2 ()'.

Listing 6.9

List of Fold : 'Load Object action'

```
{{{
INT Error :
[63]BYTE File.name :            -- Chosen filename
INT      File.name.len :        -- Length of filename

VAL X      IS  0 :
VAL Y      IS 18 :
VAL WIDTH  IS 77 :
VAL HEIGHT IS  3 :
}}}
SEQ
  draw.window  ( screen, X, Y, WIDTH, HEIGHT)
  cursor.on (from.ios, to.ios)
  get.filename ( keyboard, screen, from.filer, to.filer,
                 (X+1), (Y+1), WIDTH,
```

```
                " Enter Object Filename : ", tkf.open.read,
                File.name.len, File.name, Error)
IF
  Error = 0
    {{{  load object
    SEQ
      -- Clear screen to prepare for object slices
      to.tfg ! command; tfg.mem.preset; 0

      load.object  ( (X+1), (Y+2), WIDTH,
                     File.name.len, File.name, Error )
      IF
        Error = 0
          {{{  init parameters & read descriptor
          SEQ
            init.unit.vect ()
            init.rotate.param ()
            init.menu.tables ()

            get.string ( keyboard, screen,
                         (X+1), (Y+3), WIDTH,
                         " Descriptor string : ",
                         descriptor.len,
                         descriptor.string)
            create.top.line.2 (descriptor.len,
                               descriptor.string)
          }}}
        TRUE
          SKIP
    }}}
  TRUE
    SKIP

cursor.off (from.ios, to.ios)
screen ! tt.goto; X; Y; tt.clear.eos
```

6.2.1.4 Voxel dataset load and store

The voxel dataset has a size of DX*DY*DZ voxels, this data is distributed across NR.OF.NODES processing elements (subcube processors). The voxel dataset is represented as DZ slices of size DX*DY. Each subcube is assigned a number of slices (SCDZ). The dataset sizes on the subcube processors are named SCDX, SCDY and SCDZ. It will be clear that the following relations exist :

SCDX = DX

SCDY = DY

SCDZ = DZ / NR.OF.NODES

Loading voxel data into or out of the network is done on a slice by slice basis. Data is read from (or written to) disk in blocks, until one slice is complete, this slice is temporarily stored in the [ID][ID]BYTE image array. The slice will then be transmitted to (or has been received from) the appropriate Subcube processor. Voxel data is stored on disk as DZ slices of DY lines with DX bytes per line. Data is stored on disk in blocks of 512 bytes, so this block will contain data of more than one line if DX < 512.

Two procedures are used for I/O operations on voxeldata : Load.object () and Save.Object (). These are located in the file 'Objectio.tsr', see listing 6.10.

Reading voxel data from disk is performed by the 'Load.object ()' procedure. Each slice is read from the disk and transmitted to the network as DY*DX Bytes, preceded by the slice number. The slice number ('slice.nr') is an integer between 0 and (DZ-1). The distributors in the network will automatically decide which slices are located at a specific node and forward the slice data accordingly. This implies that the controller does not need to address a specific node when sending slices to the network, the slice.nr is sufficient. When one slice of the voxel data has been processed, the Object.load procedure will move to the next by incrementing the slice.nr counter. Each slice will be displayed during the object loading phase and a load percentage will be computed and displayed on the user console.

When the operator wishes to save a previously 'grabbed' object to disk, then this data must be requested from the subcube processors where it is located. This data is again read in from the network on a slice by slice basis. The controller does not need to address a specific node when requesting slices from the network, the slice.nr is sufficient. The distributors in the network will automatically decide which slice is located at a specific node and forward the send.slice.data command accordingly. When a node has received the command to send a slice, it will do so by sending this slice via the merger towards the controller on a line by line basis. The slice is temporarily loaded in the [ID][ID]BYTE image array and converted to [512]BYTE blocks that can be stored on disk. The described operation is performed by the Save.object procedure. The presently processed slice is being displayed on the Framegrabber board and the loading percentage is supplied on the operator console.

Both procedures perform extensive checking during disk access to prevent the system from hanging-up, should anything go wrong. This includes checking filename and filesize.

Listing : 6.10

List of File : "objectio.tsr"

File last modified : 22-11-90

```
PROC load.object (VAL INT X, Y, WIDTH,
                  INT File.name.len, [63]BYTE File.name, INT Error)

  ---  Load new Object from disk into voxel processor
  INT result :
  SEQ
    Error := 0
    open.tkf.file (from.filer, to.filer, tkf.open.read,
                   File.name.len, File.name, result)
    IF
      result=fi.ok
        {{{  Read slices and load network
        ...  declarations
        SEQ
          {{{  load object
          ...  init loading percentage
          slice.nr := 0
          WHILE (slice.nr < DZ) AND (Error = 0)
            INT y :
            SEQ
              {{{  read slice
              y := 0
              WHILE (y < DY) AND (Error = 0)
                SEQ
                  {{{  read block
                  read.tkf.block (from.filer, to.filer,
                                  length, block, result)
                  }}}
                  IF
                    result=fi.ok
                      ...  store block temporarily in image[][]
                    result=fi.eof
                      ...  Error
                    TRUE
                      ...  Error
              }}}
              IF
                (Error = 0)
                  SEQ
                    {{{  send slice to screen
                    {{{  send slice
                    to.tfg ! command; tfg.load.slice
                    SEQ y=0 FOR DY
                      to.tfg ! [ image[y] FROM 0 FOR DX ]
                    }}}
```

```
                        {{{  send top line
                        create.top.line.1 ( slice.nr )
                        to.tfg ! command; tfg.load.string;
                                 X.TOP.1; Y.TOP.1; 0; 255;
                                 (SIZE top.line.1)::top.line.1
                        }}}
                        }}}
                        {{{  send slice to subcube
                        to.net ! command; pe.load.slice; slice.nr
                        SEQ y=0 FOR DY
                          to.net ! [ image [y] FROM 0 FOR DX ]
                        }}}
                      TRUE
                        SKIP

                    slice.nr := slice.nr + 1
                }}}

                close.tkf.file (from.filer, to.filer, result)
                IF
                  result=fi.ok
                    SKIP
                  TRUE
                    ... Error
              }}}
            TRUE
              ... Error
:

PROC save.object (VAL INT X, Y, WIDTH,
                  INT File.name.len, [63]BYTE File.name, INT Error)

  --- Save Object from voxel processor onto disk
  INT result :
  INT id.len, type, content :
  [63]BYTE id :
  SEQ
    Error := 0
    make.id (from.filer, to.filer,
             [File.name FROM 0 FOR File.name.len],
             id.len, id,
             type, content, result)
    IF
      result=fi.ok
        {{{  open for write
        SEQ
          open.tkf.file (from.filer, to.filer,
                         tkf.open.write, id.len, id, result)
          IF
            result=fi.ok
              {{{  Read slices from network and write blocks
```

```
... declarations
SEQ
  {{{  save object
  ...  init loading percentage

  slice.nr := 0
  WHILE (slice.nr < DZ) AND (Error = 0)
    BOOL slice.received.ok :
    SEQ
      {{{  receive slice from network
      to.net ! command; pe.send.slice; slice.nr
      receive (from.net,
               receive.slice, slice.received.ok)
      }}}
      IF
        slice.received.ok
          {{{  store slice
          INT y :
          SEQ
            ...  send slice to screen
            ...  send top line

            y := 0
            WHILE (Error = 0) AND (y < DY)
              INT idx :
              VAL LINES.PER.SECTOR  IS  (512/DX) :
              SEQ
                {{{  fill data block with slice
                idx := 0
                SEQ i=0 FOR LINES.PER.SECTOR
                  SEQ
                    [block FROM idx FOR DX] :=
                       [image[y] FROM 0 FOR DX]
                    y := y + 1
                    idx := idx + DX
                }}}
                {{{  write block to disk
                write.tkf.block (from.filer,to.filer,
                                 512, block, result)
                }}}
                IF
                  result=fi.ok
                    ...  next block
                  TRUE
                    ...  Error
          }}}
        TRUE
          ...  Error

      slice.nr := slice.nr + 1
  }}}
```

```
                close.tkf.file (from.filer, to.filer, result)
                IF
                  result=fi.ok
                    SKIP
                  TRUE
                    ...  Error
              }}}
            TRUE
              ...  Error
        }}}
      TRUE
        ...  Error
:
```

6.2.1.5 Result images load and store

Images resulting from visualization may be stored on disk and can be retrieved for display at a later moment. These operations are performed by two procedures shown in listing 6.11, called 'imageio.tsr'. Basically, these operations are identical to those that were used for loading and storing voxeldata. The Controller allways has the latest image stored in the [ID][ID]BYTE image array. When loading an image from disk it will be read in chunks of 512 bytes which are then copied to the correct position in the image array. Error checking is performed at all relevant positions and a loading percentage is computed during operation.

Listing : 6.11

List of File : 'imageio.tsr'

File last modified : 22-11-90

```
PROC load.imag (VAL INT X, Y, WIDTH,
                INT File.name.len, [63]BYTE File.name, INT Error)
  INT result :
  SEQ
    Error := 0
    open.tkf.file (from.filer, to.filer, tkf.open.read,
                   File.name.len, File.name, result)
    IF
      result=fi.ok
        {{{  read blocks
        ...  declarations
        SEQ
          ...  init
          WHILE (result=fi.ok) AND (y < ID)
            SEQ
              {{{  read block
              read.tkf.block (from.filer,to.filer,
                              length, block, result)
              }}}
              IF
                result=fi.ok
                  {{{  store block in image[][]
                  INT idx :
                  SEQ
                    idx := 0
                    SEQ i=0 FOR (LINES.PER.SECTOR)
                      SEQ
                        image[y] := [block FROM idx FOR ID]
                        y := y + 1
                        IF
                          (y \ LOAD.STEP) = 0
                            ...  write load percentage
                          TRUE
                            SKIP
                        idx := idx + ID
                  }}}
                result=fi.eof
                  ...  Error
                TRUE
                  ...  Error
          close.tkf.file (from.filer, to.filer, result)
          IF
            result=fi.ok
              SKIP
            TRUE
```

```
              ...  Error
        }}}
      TRUE
        ...  Error
:

PROC save.imag (VAL INT X, Y, WIDTH,
                INT File.name.len, [63]BYTE File.name, INT Error)
  INT result :
  INT id.len, type, content :
  [63]BYTE id :
  SEQ
    Error := 0
    make.id (from.filer, to.filer,
             [File.name FROM 0 FOR File.name.len],
             id.len, id,
             type, content, result)
    IF
      result=fi.ok
        {{{  open for write
        SEQ
          open.tkf.file (from.filer, to.filer,
                         tkf.open.write, id.len, id, result)
          IF
            result=fi.ok
              {{{  write blocks
              ...  declarations
              SEQ
                ...  init
                WHILE (result=fi.ok) AND (y < ID)
                  INT idx :
                  SEQ
                    {{{  fill block from image[][]
                    idx := 0
                    SEQ i=0 FOR (LINES.PER.SECTOR)
                      SEQ
                        [block FROM idx FOR ID] := image[y]
                        y   := y + 1
                        idx := idx + ID
                    }}}
                    {{{  write block to disk
                    write.tkf.block (from.filer,to.filer,
                                     512, block, result)
                    }}}
                    IF
                      result=fi.ok
                        ...  next block
                      TRUE
                        ...  Error
                close.tkf.file (from.filer, to.filer, result)
                IF
```

```
                result=fi.ok
                  SKIP
                TRUE
                  ...  Error
              }}}
            TRUE
              ...  Error
          }}}
        TRUE
          ...  Error
:
```

6.2.1.6 Look-up table load and store

Listing 6.12 shows three procedures that are used to initialise and to store or load the contents of the Color Look-Up tables (LUT's). These LUT's are actually used in the Framegrabber board, but the manipulations on their contents is performed on a local copy in the Controller. This implies that the LUT's content must be transmitted to the Framegrabber after any changes.

The 'PROC init.lut ()' simply sets the contents of the four possible LUT's to certain defined values: a grey scale, an inverse grey scale and two pseudo-color scales. This data is transmitted to the Framegrabber LUT after initialisation.

The other two procedures read or write the contents of one LUT from or to disk. The LUT contents consists of three tables (Red, Green and Blue) of 256 entries each. Values from [0..63] are valid for each entry. The data is stored on disk as one file of 3*256 bytes. Some simple data manipulation is needed to convert the LUT contents, which is of integer type, to blocks of bytes. All possible file i/o errors are trapped and reported with error messages. The "PROC load.lut ()" and "PROC save.lut ()" are supplied with the selected filenames and the selected LUT. These parameters must have been checked first. Since these procedures are called from within the menu system, there user i/o takes place through the parameter window at the bottom of the screen. It is therefore necessary to supply some window parameters (X, Y, WIDTH) to the procedures, should error messages need to be displayed.

Listing 6.12

List of File : "lutio.tsr --- Init/Load/Save LUT tables"

File Last Modified : 18-12-90

```
{{{  PROC init.lut ()
PROC init.lut ()
  --- default lut values (equal to GDS values)
  INT table.nr, R, G, B:
  SEQ
    ...  table 0      --- grey scale
    ...  table 1      --- inverse grey scale
    ...  table 2      --- color scale
    {{{  table 3      --- R, G, B and combined color scales
    table.nr := 3
    SEQ
      ...    0  -   15 grey scale
      ...   16  -   32 red scale
      ...   32  -   47 green scale
      ...   48  -   63 blue scale
      ...   64  -   79 yellow scale
      ...   80  -   95 cyan scale
      ...   96  -  111 magenta scale
      ...  112  -  127 red & green scale with third blue
      ...  128  -  143 green & blue scale with third red
      ...  144  -  159 blue & redscale with third green
      ...  160  -  175 red & green scale with two-thirds blue
      ...  176  -  191 green & blue scale with two-thirds red
      ...  192  -  207 blue & red scale with two-thirds green
      ...  208  -  223 red & green scale with full blue
      ...  224  -  239 green & blue scale with full red
      ...  240  -  255 blue & red scale with full green
    }}}
:
}}}


{{{  PROC load.lut
PROC load.lut(VAL INT X, Y, WIDTH,
              INT File.name.len, [63]BYTE File.name,
              VAL INT selected.lut.nr, INT Error)
  INT result :
  SEQ
    Error := 0
    open.tkf.file (from.filer, to.filer, tkf.open.read,
                   File.name.len, File.name, result)
    IF
      result=fi.ok
        {{{  read blocks
        INT count, length:
```

```
          [512]BYTE block :           --- minimum blocksize must be 512
          SEQ
            count := 0
            WHILE (result=fi.ok) AND (count < 3)
              SEQ
                --  read block
                read.tkf.block (from.filer,to.filer,
                                length, block, result)
                IF
                  result=fi.ok
                    ...  store R and G or B block
                  result=fi.eof
                    ...  Error
                  TRUE
                    ...  Error
            close.tkf.file (from.filer, to.filer, result)
            IF
              result=fi.ok
                SKIP
              TRUE
                ...  Error
        }}}
      TRUE
        {{{  Error
        SEQ
          error.message (keyboard, screen,
                   X, Y, WIDTH,
                   " ERROR : Can not open File. Strike any key..." )
          Error := -2
        }}}
:
}}}

{{{  PROC save.lut
PROC save.lut (VAL INT X, Y, WIDTH,
               INT File.name.len, [63]BYTE File.name,
               VAL INT selected.lut.nr, INT Error)
  INT result :
  INT id.len, type, content :
  [63]BYTE id :
  SEQ
    Error := 0
    make.id (from.filer, to.filer,
             [File.name FROM 0 FOR File.name.len],
             id.len, id,
             type, content, result)
    IF
      result=fi.ok
        {{{  open for write
        SEQ
          open.tkf.file (from.filer, to.filer,
```

```
                              tkf.open.write, id.len, id, result)
                IF
                  result=fi.ok
                    {{{  write blocks
                    INT count :
                    [256]BYTE block :
                    SEQ
                      ...  write R and G or B blocks
                      close.tkf.file (from.filer,to.filer, result)
                      IF
                        result=fi.ok
                          SKIP
                        TRUE
                          ...  Error
                    }}}
                  TRUE
                    ...  Error
              }}}
            TRUE
              {{{  Error
              SEQ
                error.message (keyboard, screen,
                       X,Y,WIDTH,
                       " ERROR : Can not create File. Strike any key...")
                Error := -2
              }}}
:
}}}
```

6.2.1.7 Rotated unit vectors calculation

As explained in Chapter 2.1, we need to calculate unit steps in display space via transformation of unit vectors in object space. This operation is performed in the "PROC rotate ()". This procedure takes three user supplied rotation angles (in degrees) as input and applies the corresponding rotation matrix to the unit vectors in object space. The unit vector values are global variables called 'unit.step.??', where '??' denotes all possible combinations of x, y and z.

Listing : 6.13

List of Fold : "rotate.tsr"

```
{{{ PROC rotate ()
PROC rotate (INT X.rot.degr, Y.rot.degr, Z.rot.degr)
  VAL  DR180   IS               180.0 (REAL32) :
  REAL32       X.rot.rad, Y.rot.rad, Z.rot.rad :
  REAL32       Sin.X.rot, Cos.X.rot :
  REAL32       Sin.Y.rot, Cos.Y.rot :
  REAL32       Sin.Z.rot, Cos.Z.rot :
  SEQ
    {{{  Calculate angles in Radians
    X.rot.rad := ( (REAL32 ROUND X.rot.degr) * PI) / DR180
    Y.rot.rad := ( (REAL32 ROUND Y.rot.degr) * PI) / DR180
    Z.rot.rad := ( (REAL32 ROUND Z.rot.degr) * PI) / DR180
    }}}
    {{{  Calculate rotation factors
    Sin.X.rot := SIN (X.rot.rad)
    Cos.X.rot := COS (X.rot.rad)
    Sin.Y.rot := SIN (Y.rot.rad)
    Cos.Y.rot := COS (Y.rot.rad)
    Sin.Z.rot := SIN (Z.rot.rad)
    Cos.Z.rot := COS (Z.rot.rad)
    }}}
    {{{  Calculate unit.steps
    unit.step.xx  := (Cos.Y.rot * Cos.Z.rot)
    unit.step.xy  := (Cos.Y.rot * Sin.Z.rot)
    unit.step.xz  := -Sin.Y.rot

    unit.step.yx  := ((Sin.X.rot * Sin.Y.rot) * Cos.Z.rot) -
                      (Cos.X.rot * Sin.Z.rot)
    unit.step.yy  := ((Sin.X.rot * Sin.Y.rot) * Sin.Z.rot) +
                      (Cos.X.rot * Cos.Z.rot)
    unit.step.yz  := (Sin.X.rot * Cos.Y.rot)

    unit.step.zx  := ((Cos.X.rot * Sin.Y.rot) * Cos.Z.rot) +
                      (Sin.X.rot * Sin.Z.rot)
    unit.step.zy  := ((Cos.X.rot * Sin.Y.rot) * Sin.Z.rot) -
                      (Sin.X.rot * Cos.Z.rot)
    unit.step.zz  := (Cos.X.rot * Cos.Y.rot)
    }}}
:
}}}
```

6.2.1.8 Parameter display on Framegrabber screen

Two procedures are shown in listing 6.14 that are used to display several rendering parameters on the Framegrabber screen. The rendering parameters (mode, angles, threshold, z-cut value and grey scale) are presented by "PROC display.rot.param" and the rotated object coordinate system is drawn on the screen by "PROC display.coord.system" for reference purposes. The first procedure operates by converting the rotation parameters to text strings, which are then transmitted to the Framegrabber via a defined protocol. The coordinate system is represented by drawing scaled and projected versions of the transformed unit vectors on the screen via a defined 'draw.line' protocol. The axes are identified by printing an x, y or z next to them. These characters are printed in upper- or lower case depending on the sign of the unit vector's z direction.

Listing : 6.14

List of File : "displ.tsr"

File Last Modified : 18-12-90

```
{{{  PROC display.rot.param
PROC display.rot.param ()
  SEQ
    to.tfg ! command; tfg.load.string;
             X.TOP.2; Y.TOP.2; 0; 255; (SIZE top.line.2)::top.line.2

    create.bottom.line.1 ( X.rot.degr, Y.rot.degr, Z.rot.degr,
                           z.cut.value, threshold.value )
    to.tfg ! command; tfg.load.string;
             X.BOT.1; Y.BOT.1; 0; 255;
             (SIZE bottom.line.1)::bottom.line.1

    to.tfg ! command; tfg.grey.ref.bar

    create.bottom.line.2 ( render.mode )
    to.tfg ! command; tfg.load.string;
             X.BOT.2; Y.BOT.2; 0; 255; 16::bottom.line.2
:
}}}

{{{  PROC display.coord.system
PROC display.coord.system ()
  INT x, y :
  VAL X.CENTRE  IS  70 :
  VAL Y.CENTRE  IS 100 :
  VAL SCALER    IS 30.0(REAL32) :
  SEQ
    {{{  unit.step.x
    x := X.CENTRE + (INT TRUNC (unit.step.xx * SCALER) )
```

```
      y := Y.CENTRE + (INT TRUNC (unit.step.xy * SCALER) )

      to.tfg ! command; tfg.draw.line; X.CENTRE; Y.CENTRE; x; y

      IF
        x > X.CENTRE
          x := x + 16
        TRUE
          x := x - 32
      IF
        y > Y.CENTRE
          SKIP
        TRUE
          y := y - 16

      IF
        unit.step.xz < 0.0 (REAL32)
          to.tfg ! command; tfg.load.string; x; y; 0; 200; 1::"x"
        TRUE
          to.tfg ! command; tfg.load.string; x; y; 0; 255; 1::"X"

      }}}
      ...  unit.step.y
      ...  unit.step.z
:
}}}
```

6.2.2 The SC Autopilot

The autopilot program was written to present automatic (and continueing) demonstrations of the Voxelprocessor system. It generates sequences of keystrokes, that are interpreted by the Controller process as if they were entered from the keyboard. To provide this possibility, the Autopilot process is running in parallel with the Controller process on the root processor. The Autopilot re-routes the keyboard channel from the Host computer (the PC) to the Controller EXE. The Autopilot process can be running in two modes :

-) Inactive mode. In this case all keyboard strokes coming from the operator are simply forwarded to the Controller. This is the normal situation were the operator is in control of the Voxelprocessor. The incoming keys are checked for the occurence of certain characters, which have a special meaning. These characters are used to switch the Autopilot from Inactive- to Active mode and back.
-) Active mode. In this case, the program generates keystrokes to control an automatic demo. All operator keystrokes are blocked, except for certain keys that are used to switch back to the Inactive mode.

The overall structure of the Controller running in parallel with the Autopilot is given again in the following listing :

Listing : 6.15

List of File : 'vox_cntr.tsr'

File Last Modified : 22-11-90

```
...  link channel numbers
{{{  channels
CHAN OF ANY from.auto.pilot :

CHAN OF ANY from.tfg, to.tfg :
PLACE to.tfg    AT  linkout1 :
PLACE from.tfg  AT   linkin1 :

CHAN OF ANY from.net, to.net :
PLACE to.net    AT  linkout2 :
PLACE from.net  AT   linkin2 :
}}}

...  SC T8 auto.pilot
...  SC T8 control.tsr        --- user interface (tfg)
--  manual & autopilot control
PAR
  auto.pilot (keyboard, from.auto.pilot)
  controller (from.auto.pilot, screen,
              from.filer, to.filer, from.ios, to.ios,
              from.net, to.net, from.tfg, to.tfg)
```

The following listing (6.16) describes the structure of the Autopilot in more detail. The process is running in an endless ALT loop, checking incoming keyboard characters for special command keys. The keys are handled in the "... send user key's OR switch to auto pilot" fold (Listing 6.17). The Autopilot will switch to Active mode if the F9 key is detected. The mode is represented by the BOOLEAN variable autopilot.on. Depending on this mode, normal keystrokes will either be forwarded or blocked. The F10 key will switch the Autopilot back to inactive mode. Since the Autopilot is running in parallel with the Controller, it is necessary to explicitly stop the Autopilot if the user wishes to quit the application. This is possible by using the '@' key. The second branch of the ALT construct, the "... run autopilot" fold (Listing 6.18), will only run when the Autopilot is in its active mode. Depending on the Voxelprocessor configuration (represented by DX, DY etc.) a different demo is shown. As may be seen from the "... ct scan" fold (Listing 6.19), the

actual demo is broken up into sections, known as demo.steps. The purpose of these steps is to enable the operator to stop the demo. The keyboard channel is checked for input after each step. A demo.step consists of sending a sequence of keystrokes to the controller, seperated by time delays. At the end of a demo.step, the demo.step counter is either incremented to show the next step, or it is set back to an earlier value when the last stage has been reached (thus generating an endless loop).

Listing : 6.16
List of File : "autpilot.tsr"
File Last Modified : 22-11-90

```
PROC auto.pilot (CHAN OF INT keyboard,
                 CHAN OF ANY from.auto.pilot)
  ...  #USE
  ...  Function key definitions
  ...  Other keys
  ...  declarations
  SEQ
    running := TRUE
    auto.pilot.on := FALSE
    WHILE running
      INT ch :
      PRI ALT
        keyboard ? ch
          ...  send user key's OR switch to auto pilot
        auto.pilot.on & SKIP
          ...  run on auto pilot
:
```

Listing : 6.17
List of Fold : "... send user key's OR switch to auto pilot"

```
IF
  ch = (INT '@')
    ...  quit
  ch = F9
    ...  start auto-pilot    (Full Demo)
  ch = F10
    ...  stop auto-pilot
  auto.pilot.on
    SKIP -- eat key
  TRUE  -- auto-pilot is inactive
    from.auto.pilot ! ch
```

Listing : 6.18

List of Fold : "... run on auto pilot"

```
...  PROC's to send keystrokes
...  PROC's to delay a while
IF
  (DX = 128) AND (DZ = 128)
    ...  ct scan demo (128* 128 * 128)
  (DX = 256) AND (DZ =  32)
    ...  ic scan demo (256 * 256 * 32)
  TRUE
    auto.pilot.on := FALSE
```

Listing 6.19

List of Fold : "... ct scan demo (128 * 128 *128)"

```
-- break up the demo into steps,
-- this way the user can stop at each step.
IF
  demo.step = 0
    ...  show menu's
  demo.step = 1
    ...  load image file
  demo.step = 2
    ...  frame grabber demo
  demo.step = 3
    ...  load voxel object
  demo.step = 4
    ...  rotate data set
  demo.step = 5
    ...  interactive rotation
  demo.step = 6
    ...  more interactive rotation
  demo.step = 7
    ...  and more interactive rotation
  demo.step = 8
    ...  change rendering mode & loop to demo.step 4
  TRUE
    SKIP
```

The present implementation of the Autopilot can easily be added to or removed from any keyboard driven application. The Autopilot program has also been used in several of our Computer Graphics applications. The main disadvantage of this implementation is that all keyboard strokes (i.e. the sequence of automatically executed commands) are hardcoded in the program. This implies that every change in the demo needs a re-compilation. A better option would be to read the keys (i.e. commands) from a file. The file could be based on ASCII characters written with an editor or alternatively, the file could be generated by a logging action during normal operation. This would provide a form of 'script' file. Presently, there is no need for these improvements, but it will be considered for a future version.

6.3 The Subcube Nodes

6.3.1 The SC Node

Listing 6.20 shows the structure of the code running on all PE's. The four channels have the following functions :

-) distr.in :

   Input of commands and data by the 'distributer' from either the Controller (only for the PE connected directly to the Controller) or from the previous PE in the pipeline.

-) distr.to.next :

   The commands and data are forwarded by the 'distributer' via this channel to the next PE in the pipeline.

-) next.to.merger :

   Data, messages or quit tags from the next PE in the pipeline are received via this channel. The channel is internally connected to the 'merger' process.

-) merger.out :

   Data (i.e. merger results), messages and quit tags from this PE are send to the previous PE in the pipeline. The final destination of this data is the 'Controller'.

There are three processes running in parallel inside each PE : the 'distributer', the 'merger' and the 'subprocessor' (Fig. 5.3). The first two processes are not required for the PE at the end of the pipeline. The value 'LAST' is used to accomplish these different structures; 'LAST' is derived from the 'NODE.NR'. The 'NODE.NR' is an identification, this value is used for example to select the slices that are to be processed on the PE. The actual code of the processes running in a node is located in the library 'voxproc.tsr'.

Listing : 6.20

List of File : node.tsr

File Last Modified : 22-11-90

```
PROC node (VAL INT NODE.NR,
           CHAN OF ANY distr.in, distr.to.next,
                       merger.out, next.to.merger )
  {{{  #USE
  #USE voxconf
  #USE voxproc
  }}}
  {{{  VAL's  for processor node
  -- Node is last element of pipeline if LAST = TRUE
  VAL BOOL LAST          IS    LAST.ARRAY [NODE.NR] :
  }}}

  CHAN OF ANY distr.to.proc, proc.to.merger :
  IF
    LAST
      {{{  no distributor or merger needed
      sub.processor (NODE.NR, distr.in, merger.out)
      }}}
    TRUE
      {{{  distributor and merger
      PRI PAR
        PAR
          distributor ( NODE.NR, distr.in,
                        distr.to.proc, distr.to.next )
          merger (NODE.NR, proc.to.merger,
                  next.to.merger, merger.out)
        sub.processor (NODE.NR,
                       distr.to.proc, proc.to.merger)
      }}}
:
```

6.3.1.1 Distributer

The following listings will explain in more detail the function of the three processes within the PE's. Listing 6.21 shows the code of the 'distributer'. This process is continuously reading and interpreting tags from the distr.in channel.

Listing : 6.21

List of File : distr.tsr

File Last Modified : 22-11-90

```
PROC distributor ( VAL INT NODE.NR,
                   CHAN OF ANY distr.in, distr.to.proc,
                   distr.to.next )
  ...  #USE
  {{{  DEBUG data
  VAL ID.string  IS   "distr   " :     --- 8 BYTES
  VAL debug      IS   TRUE :
  VAL debug      IS   FALSE :
  }}}
  {{{  VAL's  for processor node
  -- Absolute number of first local slice
  VAL FIRST.SLICE      IS       (NODE.NR * SCDZ) :
  -- Absolute number of last local slice
  VAL LAST.SLICE       IS  (FIRST.SLICE + SCDZ1) :
  }}}
  BOOL running :
  SEQ
    d.write.string (distr.to.proc, debug, ID.string,
                    "distr. activated" )
    running := TRUE
    WHILE running      --- Main Loop
      BYTE tag :
      SEQ
        distr.in ? tag
        IF
          tag = command
            ...  transfer command
          tag = quit
            {{{  quit
            running := FALSE
            }}}
          TRUE
            {{{  ERROR
            e.write.string(distr.to.proc, ID.string,
               "ERROR : illegal main tag received" )
            }}}
```

```
    d.write.string(distr.to.proc, debug, ID.string,
                   "distr. killed" )
    {{{  quit
    PAR
      {{{  send to next
      distr.to.next ! quit
      }}}
      {{{  send to sub.processor
      distr.to.proc ! quit
      }}}
    }}}
:
```

Listing 6.22 shows the code of the '... transfer command' fold. Depending on the received tag either some sub tags or data is expected. Received commands and data are interpreted and forwarded to the (local) sub.processor and to the next PE, this is demonstrated for the 'pe.render' tag. That same structure is repeated for all commands, except for pe.load.slice and pe.send.slice. Slice data (i.e. voxel data) is uniquely assigned to one PE. When a new slice is transmitted to the network by the controller, each receiving PE will check wether it should store the slice locally or forward it. The slice number, together with the NODE.NR, is used to make this decision. This operation is shown in the '...load.slice' fold.

Listing : 6.22

List of Fold : transfer command

```
SEQ
  distr.in ? tag
  IF
    tag = pe.render
      {{{  render
      ...  VAR rotation parameters
      SEQ
        {{{  receive
        distr.in ? render.mode;
                   transform.vectors;
                   transform.sizes;
                   z.cut.value; threshold.value
        }}}
        PAR
          {{{  send on to next
          distr.to.next ! command; pe.render;
                          render.mode;
                          transform.vectors;
                          transform.sizes;
                          z.cut.value;
```

```
                                threshold.value
            }}}
            {{{{  send on to sub.processor
             distr.to.proc ! command; pe.render;
                             render.mode;
                             transform.vectors;
                             transform.sizes;
                             z.cut.value;
                             threshold.value
            }}}
        }}}
tag = pe.histogram
        ...  histogram
tag = pe.sobel
        ...  sobel
tag = pe.laplace
        ...  pe.laplace
tag = pe.mean
        ...  mean
tag = pe.median
        ...  median
tag = pe.grey.transform
        ...  grey.transform
tag = pe.load.slice
        {{{  load slice
        INT slice.nr :
        [SCDX]BYTE slice.line :
        SEQ
          distr.in ? slice.nr
          IF
            (slice.nr > LAST.SLICE)
              {{{  send on to next
              SEQ
                --- This can not happen
                --- for LAST nodes
                distr.to.next ! command;
                                pe.load.slice;
                                slice.nr
                SEQ y=0 FOR SCDY
                  SEQ
                    distr.in ? slice.line
                               distr.to.next ! slice.line
              }}}
            TRUE
              {{{  send on to sub.processor
              SEQ
                distr.to.proc ! command;
                                pe.load.slice;
                                slice.nr
                SEQ y=0 FOR SCDY
                  SEQ
```

```
                distr.in ? slice.line
                           distr.to.proc ! slice.line
          }}}
    }}}
  tag = pe.send.slice
    ...  send.slice
  TRUE
    ...  ERROR
```

6.3.1.2 Sub.processor

Listing 6.23 shows the code of the 'sub.processor'. This process is also continuously reading and interpreting tags from its input channel. Depending on the received tag either some sub tags or data is expected and processed accordingly.

Listing : 6.23

List of File : pe.tsr

File Last Modified : 22-11-90

```
PROC sub.processor (VAL INT NODE.NR,
                    CHAN OF ANY sub.processor.in,
                                sub.processor.out)
  ...  #USE
  ...  transf.tsr   -- procedures to perform rendering
  ...  DEBUG data
  ...  VAL's for processor node
  ...  VAR's
  {{{  PROC's
  ...  PROC mean ()
  ...  PROC median ()
  ...  PROC laplace ()
  ...  PROC sobel ()
   ..  PROC histogram ()
  ...  PROC grey.transform ()
  ...  PROC subimage.init () -- clear resulting image array
  ...  PROC voxel.limit ()   -- limit voxel index to local range
  }}}
  SEQ
    d.write.string(sub.processor.out, debug, lD.string, "pe activated" )
    ...  init
    running := TRUE
    WHILE running
      BYTE tag :
      SEQ
        sub.processor.in ? tag
```

```
        IF
          tag = command
            ...  process command
          tag = message
            ...  message
          tag = quit
            running := FALSE
          TRUE
            ...  ERROR

    d.write.string(sub.processor.out, debug, ID.string, "pe killed" )
    {{{  quit
    sub.processor.out ! quit
    }}}
:
```

The '... process command' fold is the most important part of the previous listing. All rendering and image processing operations are identified and activated from within this fold, which is shown in listing 6.24. The response to a histogram command is given as an example :

-) receive more parameters, e.g. 'threshold'.
-) convert the global indices for the volume-of-interest (running from [0..DZ> for the z index) to local indices (running from [0..SCDZ>). This implies that a PE remains inactive whenever the global indices fall outside the range assigned to this specific PE. The described operation is performed by the 'PROC voxel.limit'. The limits for the local ranges are derived from the NODE.NR.
-) compute the histogram (or any other selected function).
-) transmit the (partial) result to the 'merger'.

The other functions, like the image processing operations, are not discussed any further here. They are straightforward implementations of well known algorithms.

Listing : 6.24

List of Fold : process command

```
SEQ
  sub.processor.in ? tag
  IF
    tag = pe.render
      ...  render view on voxels
    tag = pe.histogram
      {{{  histogram
      SEQ
        {{{  receive
        sub.processor.in ? transform.sizes; threshold.value
        }}}
        {{{  initialisations
        voxel.limit ()
        }}}
        {{{  process
        histogram ()
        }}}
        {{{  send result
        sub.processor.out ! command; merge.histogram;
                            sub.histogram.data
        }}}
      }}}
    tag = pe.sobel
      {{{  sobel
      SEQ
        sobel ()
        subimage.init ()
      }}}
    tag = pe.laplace
      ...  laplace
    tag = pe.mean
      ...  mean
    tag = pe.median
      ...  median
    tag = pe.grey.transform
      ...  grey.transform
    tag = pe.load.slice
      ...  load slice
    tag = pe.send.slice
      ...  send slice
    TRUE
      ...  ERROR
}}}
```

The fold '... render view on voxels' performs the actual voxel visualisation process. It is the most important part of the visualisation software and it will be discussed in some detail now (listing 6.25).

The actual transformation from object-space to display space is performed with fixed-point REALS. These fixed-point numbers are represented with 32 bit INT's (16 bit integer, 16 bit fraction). The computed transformed normal vectors (REAL32) are converted to this new format. The advantage of this format is that the computed coordinate can easily be used as an integer index into the resulting subcube images by taking the upper 16 bits.
The transformation process must traverse the local voxel data in a 'back-to-front' ordering, i.e. the voxel that has the largest distance to the observer after the transformation is processed first. For the x-axis (and the other two axis also) this ordering is implemented by accessing the voxel data from [0..SCDX> or from <SCDX..0]. The voxel traversing routine will use a 'starting value' (idx.start.x) and an 'index step' (idx.step.x). The normal or inversed index steps ('1' or '-1') are matched by normal or inverted unit.steps for all axes. The choice of the step direction depends directly on the sign of the transformed unit vector's z-component, since this component indicates the direction from which the view on the voxel data is to be computed.
The computed transformed voxel coordinates are used for direct access of the resulting image (see Fig. 2.6). In order to achieve that the center of the transformed voxel data is at the center of the resulting image, it is sufficient to select an appropiate starting value (start.x, start.y, start.z) for the incremental coordinates. These starting values are also corrected for the offset of the local voxel data set within the projection screen.
The actual visualisation process can start when the mentioned initialisations have been made. A different algorithm is selected depending on the render mode. The render algorithms are largely equivalent, but have been coded seperately to achieve a maximum performance.

Listing : 6.25

List Of Fold : render view on voxels

```
...  VAR's
...  PROC send ()  -- send subcube image to merger
SEQ
  {{{  receive
  sub.processor.in ? render.mode;
                     transform.vectors; transform.sizes;
                     z.cut.value; threshold.value
  }}}
  {{{  initialisations
  ...  voxel.limit ()
  {{{  x direction & start.coordinates & unit.step.x
  -- initialise loop variables (voxel indices) and unit
  -- vectors to provide a 'back-to-front' transform order
  IF
    (unit.step.xz  >= 0.0 (REAL32))
      {{{  step forward through voxel data in x direction
      REAL32  start.coordinate :
      SEQ
        forward.x := TRUE
        {{{  idx.start & idx.end & idx.run
        -- ok
        }}}
        {{{  idx.step
        idx.step.x := 1
        }}}
        {{{  set start.x, start.y and start.z
        start.coordinate :=
          REAL32 ROUND (voxel.start.x - H.DX)
        start.x.r := start.coordinate * unit.step.xx
        start.y.r := start.coordinate * unit.step.xy
        start.z.r := start.coordinate * unit.step.xz
        }}}
        {{{  unit.vector.x
        --- ok
        }}}
      }}}
    TRUE
      {{{  step backwards through voxel data
      INT temp :
      REAL32  start.coordinate :
      SEQ
        forward.x := FALSE
        {{{  swap idx.start & idx.end
        temp        := idx.start.x
        idx.start.x := idx.end.x
        idx.end.x   := temp
```

```
        }}}
        {{{  invert idx.step
        idx.step.x := -1
        }}}
        {{{  set start.x, start.y and start.z
        start.coordinate :=
           REAL32 ROUND (voxel.end.x - 9.DX)
        start.x.r := start.coordinate * unit.step.xx
        start.y.r := start.coordinate * unit.step.xy
        start.z.r := start.coordinate * unit.step.xz
        }}}
        {{{  invert unit.vector.x
        unit.step.xx  := -unit.step.xx
        unit.step.xy  := -unit.step.xy
        unit.step.xz  := -unit.step.xz
        }}}
      }}}
  }}}
  ...  y direction & start.coordinates & unit.step.y
  ...  z direction & start.coordinates & unit.step.z
  ...  adapt start.coordinates to result.image coord.system
  {{{  convert transform values to fixed.point REAL
  ...  start.coordinates
  ...  unit.vector.x
  ...  unit.vector.y
  ...  unit.vector.z
  }}}

    --- flip z.cut.value
    z.cut.value := (ID - 1) - z.cut.value
    --- scaled to fixed-point value
    z.cut.i     := z.cut.value * scaler
  }}}
  {{{  process voxel data
  IF
    render.mode = VIEW
      ...  view
    render.mode = INTEGRATE
      ...  integrate
    render.mode = LAYER
      ...  layer
    render.mode = Z.SHADE
      ...  z shade
    TRUE
      ...  ERROR
  }}}
  {{{  subimage initialisation
  subimage.init ()
  }}}
}}}
```

The render algorithm for the 'front view' mode is presented in listing 6.26. The other rendering modes differ only at the deepest level, where the actual subimage value is computed. Only this part will be described explicitly for the other modes.
The basic rendering algorithm consists of three nested 'loops' that traverse the voxel data set along the z-, y-, and x-axes. For each step the transformed coordinate (xxi, xyi, xzi) is computed in an incremental way. The index values used to access the voxel data (idx.x, idx.y and idx.z) are also updated for every new step. The 'back-to-front' ordering has been provided by the values of the index steps (idx.step.x, idx.step.y, idx.step.z) and the transformed unit vectors (unit.step.xxi etc.). The idx.run.x, idx.run.y and idx.run.z values are used to restrict the number of processed voxel to the volume-of-interest. Abbreviations are used whenever possible to provide a more efficient access to the voxel data. One of these measures involves the use of four neighbouring voxels as one INT32 variable ('dummy'). These four voxels are tested for '0' and if TRUE, a jump of four voxels is made in the x direction. If one or more voxels are not zero, all of them will be individualy processed. This processing must be executed in the right back-to-front order, indicated by the pre-computed BOOL 'forward.x'.

Listing : 6.26
List of Fold : 'view'

```
SEQ
  IF
    z.cut.value = ID1
      ...  NO z.cut needed
    TRUE
      {{{  with z.cut
      SEQ
        {{{  init idx.z
        idx.z := idx.start.z
        }}}
        {{{  init zx, zy, zz
        zxi := start.xi
        zyi := start.yi
        zzi := start.zi
        }}}
        SEQ z.count=0 FOR idx.run.z
          {{{  abbreviations
          subcube.data.slice    IS subcube.data [idx.z] :
          }}}
          SEQ
            {{{  init idx.y
            idx.y := idx.start.y
```

```
}}}
{{{  init yx, yy, yz
yxi := zxi
yyi := zyi
yzi := zzi
}}}
{{{  SEQ y.count
SEQ y.count=0 FOR idx.run.y
  {{{  abbreviations
  short     IS subcube.data.slice [idx.y] :
  [SCDX >> 2]INT short.i    RETYPES short :
  }}}
  SEQ
    {{{  init idx.x
    idx.x := idx.start.x
    }}}
    {{{  init xx, xy, xz
    xxi := yxi
    xyi := yyi
    xzi := yzi
    }}}
    {{{  SEQ x.count
    ...  xxi.16, xyi.16 RETYPES
    {{{  dummy
    INT dummy :
    [4]BYTE dummy.byte RETYPES dummy :
    dummy.0          IS dummy.byte[0] :
    dummy.1          IS dummy.byte[1] :
    dummy.2          IS dummy.byte[2] :
    dummy.3          IS dummy.byte[3] :
    }}}
    subcube.image IS subcube.image :
    SEQ x.count=0 FOR idx.run.x
      SEQ
        dummy := short.i [idx.x]
        {{{  increment idx.x
        idx.x := idx.x + idx.step.x
        }}}
        IF
          dummy = 0
            {{{  skip 4 bytes
            SEQ
              {{{  increment xx, xy, xz
              xxi := xxi + unit.step.4.xxi
              xyi := xyi + unit.step.4.xyi
              xzi := xzi + unit.step.4.xzi
              }}}
            }}}
          forward.x
            {{{  process data forward
            SEQ
```

```
                    ...  dummy.0
                    ...  dummy.1
                    ...  dummy.2
                    ...  dummy.3
                    {{{  increment xx, xy, xz
                    xxi := xxi + unit.step.xxi
                    xyi := xyi + unit.step.xyi
                    xzi := xzi + unit.step.xzi
                    }}}
                  }}}
                TRUE
                  {{{  process data backward
                  SEQ
                    ...  dummy.3
                    ...  dummy.2
                    ...  dummy.1
                    ...  dummy.0
                    ...  increment xx, xy, xz
                  }}}
              }}}
              {{{  increment idx.y
              idx.y := idx.y + idx.step.y
              }}}
              {{{  increment yx, yy, yz
              yxi := yxi + unit.step.yxi
              yyi := yyi + unit.step.yyi
              yzi := yzi + unit.step.yzi
              }}}
          }}}
          {{{  increment idx.z
          idx.z := idx.z + idx.step.z
          }}}
          {{{  increment zx, zy, zz
          zxi := zxi + unit.step.zxi
          zyi := zyi + unit.step.zyi
          zzi := zzi + unit.step.zzi
          }}}
      }}}

  {{{  send result
  send ()
  }}}
}}}
```

Finally the actual processing per voxel (for all implemented rendering modes) is described in listing 6.27. A new value is computed for the subcube.image result, following a check of the voxel value and the z-distance. The indices into the subcube.image are the integer parts of the transformed fixed-point voxel coordinates (xy.i and xx.i).

Listing 6.27

```
{{{  process voxel [idx.z][idx.y][idx.x]   --- VIEW
IF
  (dummy.0 > threshold.value) AND (xzi < z.cut.i)
    subcube.image [INT xyi.16.1][INT xxi.16.1] := dummy.0
  TRUE
    SKIP
}}}


{{{  process voxel [idx.z][idx.y][idx.x]   -- INTEGRATE
IF
  (dummy.0 > threshold.value) AND (xzi < z.cut.i)
    density IS subcube.image [INT xyi.16.1][INT xxi.16.1] :
    IF
      (density < 255 (BYTE))
        density := BYTE ((INT density) + 1)
      TRUE
        density := 255 (BYTE)
  TRUE
    SKIP
}}}


{{{  process voxel [idx.z][idx.y][idx.x]   -- LAYER
IF
  (dummy.0 > threshold.value) AND (xzi < z.cut.i)
    subcube.image [INT xyi.16.1][INT xxi.16.1] :=
      layer.value
  TRUE
    SKIP
}}}


{{{  process voxel [idx.z][idx.y][idx.x]   -- Z SHADE
IF
  (dummy.0 > threshold.value) AND (xzi < z.cut.i)
    subcube.image [INT xyi.16.1][INT xxi.16.1] :=
      BYTE ((INT xzi.16.1) >> 1)
  TRUE
    SKIP
}}}
```

6.3.1.3 Merger

The third and last process within a PE is the 'merger'. This module will combine partial results from the local sub.processor with partial results from the next PE in the pipeline and trar . : he merged result to the previous PE (eventually arriving at the Controller). A second function of the merger is to simply forward data and messages from other processes towards the Controller. This structure is indicated by the two input channels ('in.0' and 'in.1') and the one output channel ('out'). The two, basically identical, channels are handled by an ALT contruct, since it is not determined which input will receive data first. The merging process is executed by the 'PROC respond ()', enabeling the use of only one piece of code for both inputs.
A BOOL parameter is used to inform this procedure which channel received input first. This input must be a 'tag', that is also supplied as a parameter.

Listing : 6.28
List of File : merger.tsr
File Last Modified : 22-11-90

```
PROC merger (VAL INT NODE.NR,
             CHAN OF ANY in.0, in.1, out)
  ---  Subcube result with lowest slice nr.
  ---  is expected via in.0
  ...  #USE
  ...  DEBUG data
  ...  VAL's  for processor node
  ...  declarations
  ...  PROC respond ()
  SEQ
    d.write.string(out, debug, ID.string,
                   "merger activated" )
    {{{  init
    nr.oi.quits := 0
    running := TRUE
    }}}
    WHILE running
      BYTE tag :
      ALT
        in.0 ? tag
          --- result slices in normal.order
          respond (in.0, in.1, out, TRUE, tag)
        in.1 ? tag
          --- result slices in inverse.order
          respond (in.1, in.0, out, FALSE, tag)

    d.write.string(out, debug, ID.string, "merger killed" )
    out ! quit
:
```

When PROC respond () is invoked, it will interpret the tag that arrived on its input.0. If this tag indicates the arrival of a message or data, then the expected packet will be received and forwarded towards the Controller. Channel 'input.1' will not be considered in this case. Listing 6.29 shows this case for 'slice.data' If however the tag indicates a 'merge command', then data from channel 'input.0' will be merged with data from 'input.1' (see listing 6.30, the fold '... merge image').
Two cases are distinguished :

-) For the view, layer and z-shade modes, a partial result will completely obscure another partial result depending on the priority. Input.0 is always given the highest priority. This means that data arriving on input.0 normally obscures data coming in from the other channel. The merging priority is however also depending on the view direction onto the voxel data. This factor is presented via the BOOL variable 'forward.merge'. The merging order (i.e. the obscuration direction) is a function of this BOOL and of the input channel priority. The obscuration function is implemented efficiently via the 'Block move' instructions of the Transputer : one image will simply overwrite the other for all pixels that are non-zero.

-) Partial results must be added up for the 'integrate' mode and for histogram computation. In this case priority is not important. The merging can not use 'block moves', but must apply basic additions per entry. Some performance improvement is achieved via writing out loop code.

The result of the merging operation will be transmitted towards the Controller on channel 'output'. As mentioned, this data may first pass other PE's for more merging operations. It is therefore necessary to include some merging parameters in the transmitted data (e.g. render mode and priority).

Listing 6.29
List of Fold : respond ()

```
PROC respond (CHAN OF ANY input.0, input.1, output,
              VAL BOOL normal.order,
              VAL BYTE tag)
  ---  This PROC will process a tag received via input.0
  ---  For the merging operation, the normal.order is :
  ---    subcube result of lower slices enters on input.0
  ---    subcube result of higher slices enters on input.1
  IF
    tag = command
      {{{  process merge command
```

```
        BYTE tag :
        SEQ
        input.0 ? tag
        IF
          tag = merge.image
            ...  merge image
          tag = merge.histogram
            ...  merge histogram
          TRUE
            ...  ERROR
        }}}
      tag = data
        {{{  process data
        BYTE tag :
        SEQ
          input.0 ? tag
          IF
            tag = slice.data
              ...  receive and forward slice
            TRUE
              ...  ERROR
        }}}
      tag = message
        ...  message
      tag = quit
        {{{  quit
        SEQ
          nr.of.quits := nr.of.quits + 1
          running := NOT (nr.of.quits = 2)
        }}}
      TRUE
        ...  ERROR
:
```

Listing : 6.30

List of Fold : merge image

```
...  declarations
SEQ
  ---  synchronise inputs, input.0 has
  ---  already received two tags
  PAR
    input.0 ? render.mode.0; forward.merge.0
    input.1 ? dummy.tag; dummy.tag;
              render.mode.1; forward.merge.1

  ---  receive & merge results
  SEQ
    output ! command; merge.image;
             render.mode.0; forward.merge.0
```

```
--- The merger needs the merge.order and
--- the slice.order
IF
  (forward.merge.0 AND normal.order) OR
    ((NOT forward.merge.0) AND
      (NOT normal.order))
    ...  result.1 obscures result.0
  TRUE
    ...  result.0 obscures result.1
```

6.4 The Graphics System

The Framegrabbber software provides access to a video-memory of 512*1024 Bytes. Data written in this memory is directly displayed on a monitor, using a color look-up table for the conversion into RGB values. Alternatively, the video memory can be filled by digitizing an analogue video input signal (e.g. from a camera). This digitized data can be accessed by the software and may be used for further processing. Given the fact that the framegrabber software plays only a supporting role in the voxel processor, it will not be discussed in great detail here. Only the general structure will be explained.

Listing 6.31 shows the main part of the framegrabber code. First some initialisations are performed :

-) The timing of the video controller hardware is set to CCIR values and the generation of an image is enabled. As a result of the CCIR norm, only a 512*760 section of the total video-memory will be visible on the screen. These procedures are rather 'close' to the TFG hardware and are largely based on manufacturer software.
-) The contents of the color look-up tables is initialised. This data may be overwritten later under command of the Controller.
-) The video-memory contents is cleared, resulting in a blanked screen.

After initialising the hardware the two main modules, running in parallel, are activated :

-) The PROC framegrabber ()

   This procedure, provided by the board manufacturer, may be accessed across the channel 'grab'. Sending commands over this channel enables or disables the acquisition of digitized images. The framegrabber will either continuously display the input data or 'freeze' the last frame. The video memory is not available for software accesses (e.g. to load a new image) if the framegrabber is active, this is indicated by the BOOL 'display.available'.

-) The '... command interpreter' fold

The program is running in an endless loop within this fold, waiting for new commands from the Controller. Commands may trigger a framegrabbing action or may act upon the video-memory directly (e.g. to draw a line).

Listing : 6.31

List of File : tfg.tsr

File last Modified : 22-11-90

```
PROC tfg.graph (CHAN OF ANY tfg.in, tfg.out, net.loader)
  ... DEBUG data
  ... #USE
  ... declarations
  ... procedures
  SEQ
    d.write.string(tfg.out, debug, ID.string,
                   "grabber is activated" )
    ...  init tfg
    PRI PAR
      frame.grabber ( grab )
      ...  command interpreter

    d.write.string(tfg.out, debug, ID.string,
                   "grabber killed" )
    tfg.out ! quit
:
```

The fold '... command interpreter' is shown in more detail in Listing 6.32. It's main purpose is to receive and execute the tfg commands that were explained before. Most of these commands involve very straighforward access to the video-memory, for example :

-) copying a received image to the memory, resulting in the display of that image (or slice)

-) sending (part of) the video-memory contents to the Controller for further processing, used to grab new voxel slices.

-) filling an area of the memory with a constant value, to clear or initialise the screen

Some functions require additional processing like drawing a line between two coordinates. These functions are well known graphics operations (e.g. the Bresenham algorithm) and are not explained here.

Other operations are :

-) draw a polygon
-) draw a circle
-) draw a character from a bitmap table
-) fill an area enclosed by a polygon with a color
-) draw a test screen or display a company logo
-) convert an array of data into a plot of this data on the screen, e.g. a histogr.

The remaining operations are used to access the color look-up tables by changing a single entry or loading a completely new contents.

Listing : 6.32

List of Fold : command interpreter

```
{{{ command interpreter
...  declarations
SEQ
  running := TRUE
  --- Wait until SC frame.grabber is initialized
  display.available := FALSE
  WHILE running
    BYTE tag :
    SEQ
      tfg.in ? tag
      IF
        tag = command
          ...  process command
        tag = quit
          ...  quit
        TRUE
          ...  ERROR
}}}
```

6.5 The network configuration

Listing 6.33 gives an overview of the network configuration file. There are three types of programs running in the network nodes :

-) The code for the framegrabber/display unit
-) The code for a subprocessor node
-) The code for the subprocessor that is directly connected to the EXE controller. This code is internally identical to a normal subprocessor node, but is has a separate dummy channel ('net.loader') to provide a connection with the framegrabber node. This channel is necessary to provide a boot path for the Transputer Development system from the framegrabber to the Subprocessors.

The listing shows a very straightforward implementation of a pipeline with 16 subprocessors (shown in Fig. 5.1), connected by two pairs of channels (command.pipe and merge.pipe). The number of processors is a constant that can easily be changed. This does however require a change in the libraries and a recompilation of the SC's also.

Listing : 6.33
List of File : vox_net.tsr
File last Modified : 22-11-90

```
{{{  PROGRAM vox_net.tsr
...  SC T8 tfg.tsr           --- Framegrabber /Display
...  SC T8 node.tsr          --- Sub Processor Node
...  SC T8 exenode.tsr       --- Sub Processor Node
                             --- connected to EXE
...  link channel numbers
{{{  channels
CHAN OF ANY tfg.in, tfg.out, net.loader :
CHAN OF ANY to.net, from.net :
[20]CHAN OF ANY command.pipe, merge.pipe :
}}}

PLACED PAR
  VAL node.nr  IS 0 :
  PROCESSOR node.nr T8
    {{{  node 0
    PLACE to.net                AT linkin0   :
    PLACE from.net              AT linkout0  :
    PLACE command.pipe [node.nr] AT linkout2  :
    PLACE merge.pipe   [node.nr] AT linkin2   :
    PLACE net.loader            AT linkout1  :
```

```
  exe.node (node.nr, to.net, command.pipe [node.nr],
                     from.net, merge.pipe [node.nr],
                     net.loader )
  }}}

PLACED PAR node.nr = 1 FOR 15
  PROCESSOR node.nr T8
    {{{  node nr
    PLACE command.pipe [node.nr-1] AT linkin0  :
    PLACE merge.pipe   [node.nr-1] AT linkout0 :
    PLACE command.pipe [node.nr]   AT linkout2 :
    PLACE merge.pipe   [node.nr]   AT linkin2  :
    node  (node.nr, command.pipe [node.nr-1],
                    command.pipe [node.nr],
                    merge.pipe [node.nr-1],
                    merge.pipe [node.nr] )
    }}}

 PROCESSOR 100 T8
   {{{  tfg
   PLACE tfg.in  AT linkin0    :
   PLACE tfg.out AT linkout0   :
   PLACE net.loader AT linkin2 :
   tfg.graph (tfg.in, tfg.out, net.loader)
   }}}
}}}
```

7 PERFORMANCE

Typical rendering speeds on a 16 Transputer system are 1 sec. for 2 Mbyte (128*128*128) voxel-images. In the table below a comparison is made for different types of operations and different numbers of processors (timing in seconds).

Nodes	2	4	8	16
view	3.7	2.4	1.4	0.9
integrate	4.2	2.7	1.6	1.0
layer	3.6	2.3	1.3	0.9
shade	4.5	2.8	1.6	1.0
mean	31.9	16.5	9.4	5.9
median	95.7	52.5	28.3	15.5
edges	65.0	33.2	17.3	8.8
convolution	103.8	52.6	27.0	17.0
histogram	2.8	1.5	1.0	0.5

The visualization operations have a communication overhead for the transmission of the resulting images to the controller of about 0.3 sec. This overhead is constant for any number of processors, so it explains the non-linear performance increase from 2 to 16 processors. It is obvious that a further increase in the number of processors would not be very useful for the given problem size : there should always be a good balance between communication- and computational demand. The voxel processor system could however achieve the same speed (about 1 sec. per image) on much larger data sizes using more Transputers. As stated before, CT images are of a larger size (typically 512*512*256) and our system could become a very effective visualization machine for this type of data also.

The 3D image processing operations would still benefit from adding more Transputers, even for the given data size, since these problems are still computational bound with 16 processors. However, for a certain number of processors we would also find a reduced efficiency here.

Some resulting images of the voxel processor are shown here as (screen) photographs :

a) Integrated Circuit scanned with the CLSM, resolution 256*256*32.
   -) Front view, (Photo 7.1)
   -) Layer view with z-cut (Photo 7.2)

b) CLSM scan of a biological object (pollen), resolution 256*256*32.
   -) Front view with grey-scale transform to increase visibility (Photo 7.3)

c) CT scan of a baby head, provided by Philips Medical Systems. Resolution 128*128*128.
   -) Front with z-cut (Photo 7.4)
   -) Integrate mode with threshold (Photo 7.5)

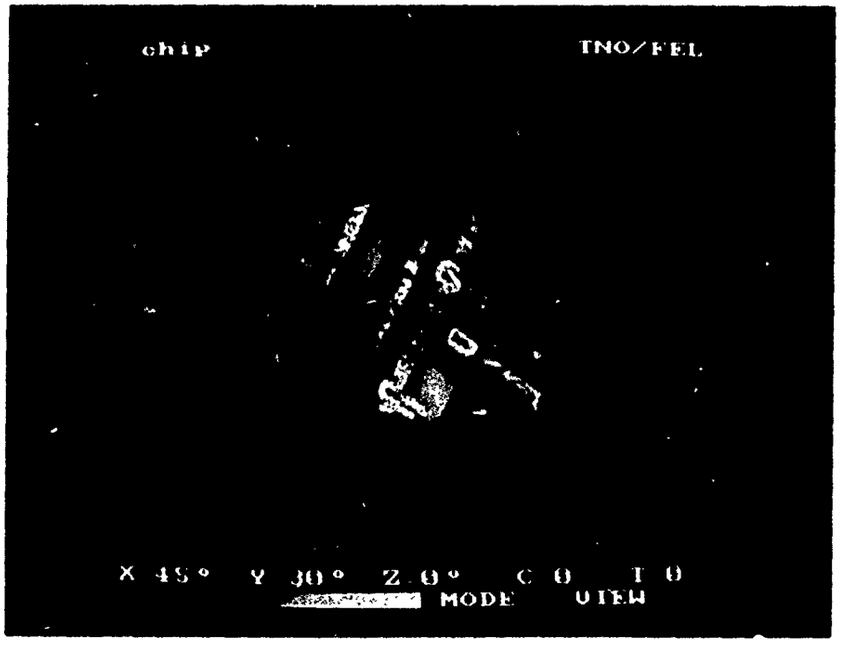


Photo 7.1 Front view

Photo 7.2 Layer view with z-cut

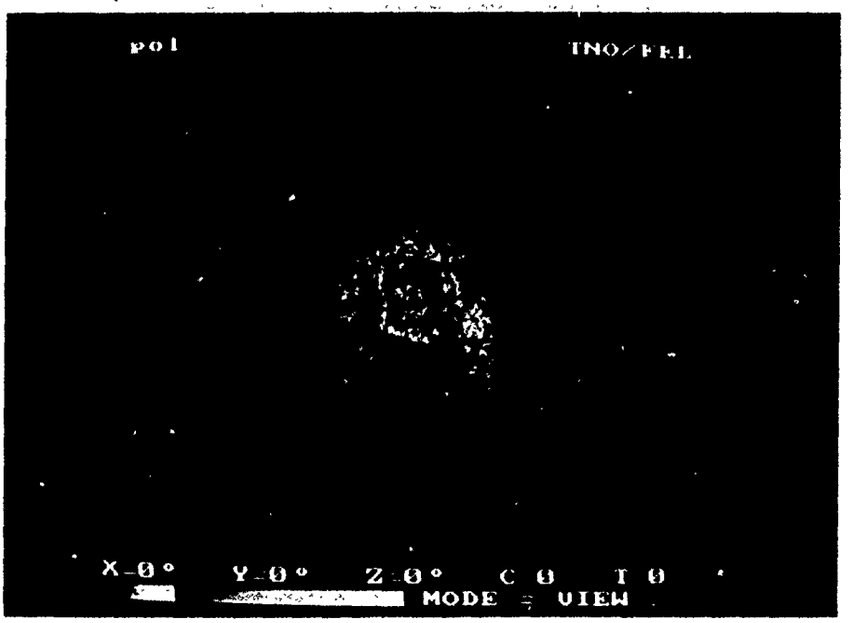


Photo 7.3 Front view with grey-scale transform to increase visibility

Photo 7.4 Front with z-cut

Photo 7.5 Integrate mode with threshold

8 FUTURE ACTIVITIES

The CLSM is developed into a commercial product by TRACOR Northern (USA). Researchers in confocal microscopy and other areas have attended presentations of the voxel processor prototype and showed an interest in the technology. The development of the prototype into a product will require a further improvement of rendering algorithms and added functionality :

a) Addition of more 3D image-processing algorithms. An important feature will be the computer assisted image segmentation (region growing) to select interesting areas in the voxel-image. This option will need communication between neighbour nodes. Region growing is a difficult task that has not been generally solved for 2D data. An implementation on 3D data will require much research.

b) Implementation of 3D geometrical measurements. For medical- and biological- imaging geometrical data is very important. Surface computations, distances and volume measurements have to be applied to the objects in the voxel-space.

c) Implementation of improved rendering algorithms, including perspective projection and shaded views.

d) Increase system performance by further code improvement and architecture optimization. For larger voxel-data sizes a system with 32 or more Transputers could be used. In this larger system the architecture will be changed to a tree structure. The advantage of a tree over a pipeline is the shorter average length of the communication path between the PE's and the Controller. The tree architecture would not require a large effort to program, since the basic structure of the software could still be used. Figure 8.1 shows an eight processor version for the network, with one slice assigned to each node. The Distributer, Subprocessor and Merger processes of the pipeline version may be used here without change. The difference lies in the addition of an extra Distributer/Merger pair. Every Distributer will select which slices must be processed locally and which must be forwarded. The Decision depends solely on the Node.Nr, as is indicated in Fig. 8.1. The Mergers will combine the partial results correctly if the provided data is in the right geometrical order.

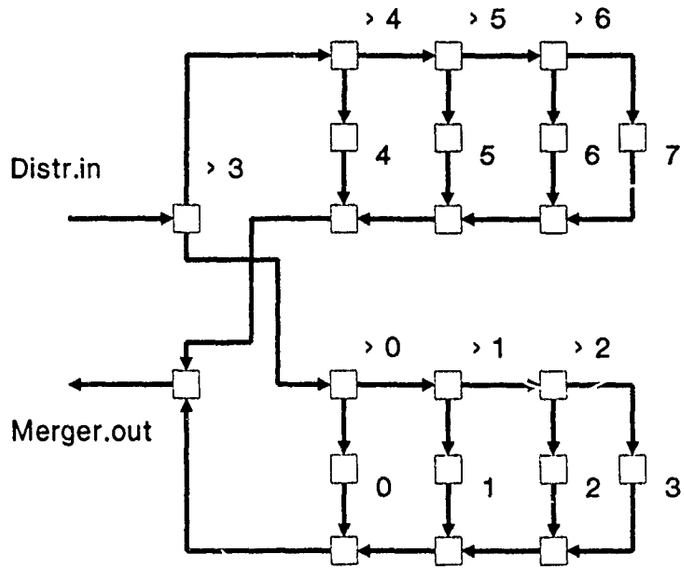


Fig. 8.1: Tree Architecture with slice distribution scheme

e) In the present system, Transputer links are used for all inter-processor communication. The link bandwidth of 1.5 MByte/s may become a bottle-neck in a future version, when slice data has to be exchanged between neighbours. A possible solution would be to use dual-port memory connected to buses for data transfer between nodes. In this case the links could supply all necessary synchronization between nodes. It would be preferable if such a data-exchange bus conforms to an industry standard.

f) Investigate (voxel) data-compression, determine effects on data transport times and implications on transform algorithms.

g) Feasibility study on stereoscopic display facilities. This option is interesting for several applications in medical- and biological research. Basically the voxel processor would have to create two images of the same object from slighly different angles. Both images would be presented on a single display to an operator that must wear a special type of spectacles.

h) Integration of the voxel processor with the CLSM (or any other sensor) requires the full control the scanner operation from within the system.

i) The present version of the voxelprocessor is implemented as a dedicated system, running code that is only suitable for Transputers. This OCCAM code does supply the highest possible performance, but it is not easily portable. An option that is worth considering is the implementation of the software on a system running under the HELIOS Operating System. HELIOS is a recent development that is quickly becoming an industry standard OS for Transputers. It provides a UNIX like 'look and feel'. Programming under HELIOS is in 'C' which has a far higher exceptance level (notably in the USA) than OCCAM. Another advantage of 'C' is the possibility to integrate existing code from other image processing packages into the system. HELIOS can use PC's and SUN workstations as a host computer. An excellent graphics output and a menu controlled user-interface can be provided via calls to the HELIOS supported X-window library. The graphics output is either directed to a dedicated Transputer board or to the SUN system. HELIOS also supports fileservers running on Transputer based hardware (e.g. harddisks, tapestreamers). The use of this hardware would remove the i/o interface bottleneck to the host computer. The performance loss of perhaps 50 % over OCCAM code may be well worth paying, considering the advantages of HELIOS for non real-time applications.

9 CONCLUSIONS

The potential of parallel processing for volume rendering applications has been clearly demonstrated by this project. Transputers have proved to be a very powerful tool, both for research and applications. The development of the system software was greatly simplified by the clear representation and support of parallelism that OCCAM offers [16]. The following conclusion can be drawn from the results of this project :

-) The interactive performance with all the required prototype functionality can be delivered by a 16 Transputer system.
-) System scalability is good if the size of the dataset is not to small. In practise, a 1 sec. response time will be achievable even for large voxel datasets. Higher performance rates at a lower cost will soon be possible with the new generation of Transputers (T9000 series).
-) The voxel processor is a high-performance, low-cost and small-sized system. The developed software is highly modular and easily adaptable.
-) The prototype is a general purpose software framework for 3D image processing. Image processing in 3D is however only considered useful in combination with a visualization tool.
-) Several alternative hardware configurations of the system are possible :
    -) Standalone version.
    -) System hosted by a PC-AT.
    -) Accelerator connected to a (SUN) Workstation.

    The hardware is commercially available from several vendors.
-) Transputers are general purpose processors and the voxel processor hardware may also be used for other (computational intensive) applications. This may require a software configurable topology which is also commercially available.
-) The voxel processor system is not limited to CLSM images only, other sources of data are equally suitable (e.g. CT scans). The system will be brought to the attention of potential users in these other areas. An important feature of the system is the flexibilty to changes in resolution, performance and rendering algorithms (very linear cost/performance function). It will therefore be very well possible to adapt the system to different application fields.
-) The development of the prototype into a product will require a further improvement of rendering algorithms and the addition of more functionality. Any further developments must be targeted at a specific application.

10 REFERENCES

[1] A. Draaijer, P.M. Houpt.
A Standard Video-rate Confocal Laser Scanning Reflection and Fluorescence Microscope. Scanning Vol. 10, 1989.

[2] S. M. Goldwasser and R. A. Reynolds.
Real-Time Display and Manipulation of 3D Medical Objects : The Voxel Processor Architecture. Computer Vision, Graphics and Image Processing 39. 1-27 (1987).

[3] A. Kaufman and R. Bakalash.
Memory and Processing Architecture for 3-D Voxel-Based Imagery. IEEE Computer Graphics & Applications. November 1988.

[4] A. C. Tan, R. Richards, A.D. Linney.
3-D Medical Graphics - Using the T800 Transputer. Proceedings of the 8th technical meeting of the OCCAM User Group. March 1988.

[5] B. Furht.
A Contribution to Classification and Evaluation of Structures for Parallel Computers. Microprocessing and Microprogramming 25, 1989.

[6] The Transputer Databook. INMOS publication, 1989.

[7] C.A.R. Hoare (Ed).
OCCAM 2 Reference Manual, Prentice Hall, 1988.

[8] W.M. ter Kuile, P. Zandveld, A. Draayer.
Beeldverwerking voor confocaal LASER scan microscoop.
TNO-MT rapport R89/300, September 1989.

[9] J.G. Harp, K.J. Palmer, H.C. Webber.
Image Processing on the Reconfigurable Transputer Processor. Proceedings of the 7th technical meeting of the OCCAM User Group. September 1987.

[10] R.S. Cok.
A medium grained parallel computer for image processing. Proceedings of the 8th technical meeting of the OCCAM User Group. March 1988.

[11] A. Rosenfeld and A.C. Kak.
Digital Picture Processing. New York, Academic press, 1976.

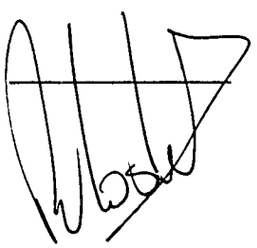
[12] MTM-2 Multi Transputer Module, Technical documentation, version 1.3. PARSYTEC GmbH, July 1987.

[13] TFG Transputer Frame Grabber, Technical documentation, version 1.1. PARSYTEC GmbH, June 1988.

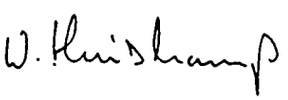
[14] TPM-4 Transputer Processor module, Technical documentation, version 1.0. PARSYTEC GmbH, July 1987.

[15] MULTITOOL 5.0 Manual, PARSYTEC publication, 1989.

[16] W. Huiskamp, P.L.J. van Lieshout et al.
Visualization of 3D Empirical Data : The Voxel Processor. Proceedings of the 10th technical meeting of the OCCAM User Group. April 1989.



Ir. P.L.J. van Lieshout
(group leader)



Ir. W. Huiskamp
(author/ project leader)

UNCLASSIFIED

p. 119

REPORT DOCUMENTATION PAGE (MOD-NL)

1. DEFENSE REPORT NUMBER (MOD-NL)	2. RECIPIENT'S ACCESSION NUMBER	3. PERFORMING ORGANIZATION REPORT NUMBER
TD91-2620		FEL-91-B166

4. PROJECT/TASK/WORK UNIT NO.	5. CONTRACT NUMBER	6. REPORT DATE
20449	—	JUNE 1991

7. NUMBER OF PAGES	8. NUMBER OF REFERENCES	9. TYPE OF REPORT AND DATES COVERED
119 (INCL. RDP, EXCL. DISTRIBUTION LIST)	16	FINAL REPORT

10. TITLE AND SUBTITLE
VOXEL DATA PROCESSING ON A TRANSPUTER NETWORK

11. AUTHOR(S)
IR. W. HUISKAMP

12. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TNO PHYSICS AND ELECTRONICS LABORATORY, P.O. BOX 96864, 2509 JG THE HAGUE
OUDE WAALSDORPERWEG 63, THE HAGUE, THE NETHERLANDS

13. SPONSORING/MONITORING AGENCY NAME(S)
TNO DIVISION OF NATIONAL DEFENSE RESEARCH, THE NETHERLANDS

14. SUPPLEMENTARY NOTES

15. ABSTRACT (MAXIMUM 200 WORDS, 1044 POSITIONS)

WITH THE GROWING AVAILABILITY OF 3D SCANNING DEVICES LIKE COMPUTER TOMOGRAPHS (CT) OR CONFOCAL LASER SCANNING MICROSCOPES (CLSM) THE NEED FOR HIGH PERFORMANCE VOLUME DATA (VOXEL) PROCESSING AND DISPLAY SYSTEMS INCREASED ENORMOUSLY. THE RECENT DEVELOPMENT OF A FAST CLSM BY IMW-TNO REQUIRED A VISUALISATION TOOL OF MATCHING PERFORMANCE. FEL-TNO WAS INVOLVED IN THE CLSM PROJECT BECAUSE OF ITS EXPERTISE IN THE AREA OF FAST VISUALIZATION TECHNIQUES USING PARALLEL PROCESSING. THE FEL-TNO TASK IN THE PROJECT WAS TO DEVELOP AN EXPERIMENTAL SYSTEM THAT DEMONSTRATES THE POTENTIAL OF PARALLEL PROCESSING FOR VOLUME RENDERING APPLICATIONS. THIS REPORT DESCRIBES THE DEVELOPMENT, IMPLEMENTATION AND EVALUATION OF THE PROTOTYPE 3D IMAGE PROCESSING SYSTEM. TOPICS OF THE REPORT ARE :

- INTRODUCTION ON VOLUME DATA PROCESSING;
- INTRODUCTION ON TRANSPUTERS AND PARALLEL PROCESSING;
- DESIGN OF THE TRANSPUTER BASED VOXEL PROCESSING SYSTEM;
- IMPLEMENTATION OF PARALLEL VOXEL VISUALIZATION;
- IMPLEMENTATION OF PARALLEL IMAGE PROCESSING ALGORITHMS;
- DETAILED DESCRIPTION OF THE SOFTWARE;
- PERFORMANCE EVALUATION AND SCALABILITY
- FUTURE DEVELOPMENTS.

16. DESCRIPTORS	IDENTIFIERS
PARALLEL PROCESSING,	IMAGE GENERATION,
DATA AND DISPLAY SYSTEM,	IMAGE PROCESSING,
* algorithms,	VOLUME RENDERING,
	* TRANSPUTERS,

17a. SECURITY CLASSIFICATION (OF REPORT)	17b. SECURITY CLASSIFICATION (OF PAGE)	17c. SECURITY CLASSIFICATION (OF ABSTRACT)
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

18. DISTRIBUTION/AVAILABILITY STATEMENT	17d. SECURITY CLASSIFICATION (OF TITLES)
UNLIMITED AVAILABILITY	UNCLASSIFIED

UNCLASSIFIED